# A Geometric Algorithm for Scalable Multiple Kernel Learning

**John Moeller**
moeller@cs.utah.edu
University of Utah

**Parasaran Raman**
praman@yahoo-inc.com
Yahoo

**Avishek Saha**
avishek2@yahoo-inc.com
Yahoo Labs

**Suresh Venkatasubramanian**
suresh@cs.utah.edu
University of Utah

## Abstract

We present a geometric formulation of the Multiple Kernel Learning (MKL) problem. To do so, we reinterpret the problem of learning kernel weights as searching for a kernel that maximizes the minimum (kernel) distance between two convex polytopes. This interpretation combined with novel structural insights from our geometric formulation allows us to reduce the MKL problem to a simple optimization routine that yields provable convergence as well as quality guarantees. As a result our method scales efficiently to much larger data sets than most prior methods can handle. Empirical evaluation on eleven datasets shows that we are significantly faster and even compare favorably with a uniform unweighted combination of kernels.

## 1 Introduction

Multiple kernel learning is a principled alternative to choosing kernels (or kernel weights) and has been successfully applied to a wide variety of learning tasks and domains [18, 4, 2, 36, 10, 35, 22, 26]. Pioneering work by Lanckriet et al. [18] jointly optimizes the Support Vector Machine (SVM) task and the choice of kernels by exploiting convex optimization at the heart of both problems. Although theoretically elegant, this approach requires repeated invocations of semidefinite solvers. Other existing methods [26, 18, 25, 32, 33], albeit accurate, are slow and have large memory footprints.

In this paper, we present an alternate *geometric* perspective on the MKL problem. The starting point for our approach is to view the MKL problem as an optimization of kernel distances over convex polytopes. The ensuing formulation is a Quadratically Constrained Quadratic Program (QCQP)

which we solve using a novel variant of the Matrix Multiplicative Weight Update (MMWU) method of Arora & Kale [3]; a primal-dual combinatorial algorithm for solving Semidefinite Programs (SDP) and QCQPs. While the MMWU approach in its generic form does not yield an efficient solution for our problem, we show that a careful geometric reexamination of the primal-dual algorithm reveals a simple alternating optimization with extremely light-weight update steps. This algorithm can be described as simply as: "find a few violating support vectors with respect to the current kernel estimate, and reweight the kernels based on these support vectors".

Our approach (a) does not require commercial cone or SDP solvers, (b) does not make explicit calls to SVM libraries (unlike alternating optimization based methods), (c) provably converges in a fixed number of iterations, and (d) has an extremely light memory footprint. Moreover, our focus is on optimizing MKL on a *single machine*. Existing techniques [26] that use careful engineering to parallelize MKL optimizations in order to scale can be viewed as complementary to our work. Indeed, our future work is focused on adding parallel components to our already fast optimization method.

A detailed evaluation on eleven datasets shows that our proposed algorithm (a) is fast, even as the data size increases beyond a few thousand points, (b) compares favorably with LibLinear [11] after Nyström kernel approximations are applied as feature transformations, and (c) compares favorably with the *uniform* heuristic that merely averages all kernels without searching for an optimal combination. As has been noted [7], the *uniform* heuristic is a strong baseline for the evaluation of MKL methods. We use LibLinear with Nyström approximations (LIBLINEAR+) as an additional scalable baseline, and we are able to beat both these baselines when both $m$ and $n$ are significantly large.

## 2 Related Work

In practice, since the space of all kernels can be unwieldy, many methods operate by fixing a base set of kernels and determining an optimal (conic) combination. An early approach (UNIFORM) eliminated the search and simply used

an equal-weight sum of kernel functions [22]. In their seminal work, Lanckriet et al. [18] proposed to simultaneously train an SVM as well as learn a convex combination of kernel functions. The key contribution was to frame the learning problem as an optimization over positive semidefinite kernel matrices which in turn reduces to a QCQP. . Soon after, Bach et al. [4] proposed a block-norm regularization method based on *second order cone programming* (SOCP).

For efficiency, researchers started using alternating optimization methods that alternate between updating the classifier parameters and the kernel weights. Sonnenburg et al. [26] modeled the MKL objective as a cutting plane problem and solved for kernel weights using Semi-Infinite Linear Programming (SILP) techniques. Rakotomamonjy et al. [25] used sub-gradient descent based methods to solve the MKL problem. An improved level set based method that combines cutting plane models with projection to level sets was proposed by Xu et al. [32]. Xu et al. [33] also derived a variant of the equivalence between group LASSO and the MKL formulation that leads to closed-form updates for kernel weights. However, as pointed out in [7], most of these methods do not compare favorably (both in accuracy as well as speed) even with the simple *uniform* heuristic.

Other works in MKL literature study the use of different kernel families, such as Gaussian families [19], hyperkernels [20] and non-linear families [29, 8]. Regularization based on the $\ell_2$-norm [16] and $\ell_p$-norm [15, 30] have also been introduced. In addition, stochastic gradient descent based online algorithms for MKL have been studied in [21]. Another work by Jain et al. [13] discusses a scalable MKL algorithm for dynamic kernels. We briefly discuss and compare with this work when presenting empirical results (Section 5).

In *two-stage kernel learning*, instead of combining the optimization of kernel weights as well as that of the best hypothesis in a single cost function, the goal is to learn the kernel weights in the first stage and then use it to learn the best classifier in the second stage. Recent two-stage approaches seem to do well in terms of accuracy – such as Cortes et al. [9], who optimize the kernel weights in the first stage and learn a standard SVM in the second stage, and Kumar et al. [17], who train on meta-examples derived from kernel combinations on the ground examples. In Cortes et al. [9], the authors observe that their algorithm reduces to solving a meta-SVM which can be solved using standard off-the-shelf SVM tools such as LibSVM. However, despite being highly efficient on few examples, LibSVM is very inefficient on more than a few thousand examples due to quadratic scaling [6]. As for Kumar et al. [17], the construction of meta-examples scales quadratically in the number of samples and so their algorithm may not scale well past the small datasets evaluated in their work.

Interestingly, our proposed MWUMKL can easily be run as a *single*-kernel algorithm. We can then apply our scalability to the two-stage algorithm of [9], allowing it not to be limited by the same constraints as LibSVM (which scales quadratically or worse in the number of examples [6]).

## 3 Background

**Notation.** We will denote vectors by boldface lower case letters like $\mathbf{z}$, and matrices by bold uppercase letters $\mathbf{M}$.

| | |
|---|---|
| $\mathbf{0}$ | zero vector or matrix |
| $\mathbf{1}$ | all-ones vector or matrix |
| $\mathbf{M} \succeq 0$ | $\mathbf{M}$ is positive semidefinite |
| $\mathbf{A} \bullet \mathbf{B}$ | $\mathrm{Tr}(\mathbf{AB}) = \sum_{i,j} A_{ij} B_{ij}$ |
| $\mathrm{diag}(\mathbf{a})$ | The diagonal matrix $\mathbf{A}$ such that $A_{ii} = a_i$ |

**Modeling the geometry of SVM.** Suppose that $\mathbf{X} \in \mathbb{R}^{n \times d}$ is a collection of $n$ training samples in a $d$-dimensional vector space (the rows $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$ are the points). Also, $\mathbf{y} = (y_1, y_2, \ldots, y_n) \in \{-1, +1\}^n$ are the binary class labels for the data points in $\mathbf{X}$. Let $\mathbf{X}_+ \subset \mathbf{X}$ denote the rows corresponding to the positive entries of $\mathbf{y}$, and likewise $\mathbf{X}_- \subset \mathbf{X}$ for the negative entries.
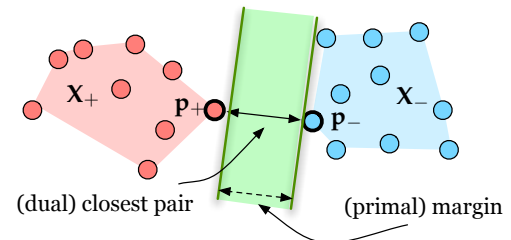


Figure 1: Illustration of primal-dual relationship for classification.

From standard duality, the maximum margin SVM problem is equivalent to *finding the shortest distance between the convex hulls of $\mathbf{X}_+$ and $\mathbf{X}_-$*. This shortest distance between the hulls will exist between two points on the respective hulls (see Figure 1). Since these points are in the hulls, they can be expressed as some convex combination of the rows of $\mathbf{X}_+$ and $\mathbf{X}_-$, respectively. That is, if $\mathbf{p}_+$ is the closest point on the positive hull, then $\mathbf{p}_+$ can be expressed as $\alpha_+^\top \mathbf{X}_+$, where $\alpha_+^\top \mathbf{1} = 1$ and $\alpha_j \geq 0$, with a similar construction for $\mathbf{p}_-$ and $\alpha_-$.

This in turn can be written as an optimization

$$\min_{\alpha} \frac{1}{2} \|\mathbf{p}_+ - \mathbf{p}_-\|^2 \qquad (3.1)$$
$$\text{s.t.} \quad \alpha_+^\top \mathbf{1} = 1, \quad \alpha_-^\top \mathbf{1} = 1, \quad \alpha_+, \alpha_- \geq 0$$

Collecting all the $\alpha$ terms together by defining $\alpha_j \triangleq \alpha_{y_j, j}$, and expanding the distance term $\|\mathbf{p}_+ - \mathbf{p}_-\|^2$, it is straight-

forward to show that Problem (3.1) is equivalent to

$$\min_{\alpha} \frac{1}{2}\alpha^{\top}\mathbf{Y}\mathbf{X}\mathbf{X}^{\top}\mathbf{Y}\alpha - \alpha^{\top}\mathbf{1} \tag{3.2}$$
$$\text{s.t.} \quad \alpha^{\top}\mathbf{y} = 0, \quad \alpha_i \geq 0.$$

where $\alpha^{\top}\mathbf{Y}\mathbf{X}\mathbf{X}^{\top}\mathbf{Y}\alpha$ is merely a compact way of writing $\sum_{j,k\in X}\alpha_j\alpha_k y_j y_k \langle\mathbf{x}_j,\mathbf{x}_k\rangle$. Problem (3.2) is of course the familiar dual SVM problem. The equivalence of (3.1) and (3.2) is well known, so we decline to prove it here; see Bennett & Bredensteiner [5] for a proof of this equivalence.

**Kernelizing the dual.** The geometric interpretation of the dual does not change when the examples are transformed by a *reproducing kernel Hilbert space* (RKHS). The Euclidean norm of the base vector space in $\|\mathbf{p}_+ - \mathbf{p}_-\|^2$ is merely substituted with the RKHS norm:

$$\|\mathbf{p}_+ - \mathbf{p}_-\|_\kappa^2 = \kappa(\mathbf{p}_+,\mathbf{p}_+) + \kappa(\mathbf{p}_-,\mathbf{p}_-) - 2\kappa(\mathbf{p}_+,\mathbf{p}_-),$$

where the *kernel function* $\kappa$ stands in for the inner product. This is dubbed the *kernel distance* [24] or the *maximum mean discrepancy* [12]. The dual formulation then changes slightly, with the covariance term $\mathbf{X}\mathbf{X}^{\top}$ being replaced by the kernel matrix $\mathbf{K}$. For brevity, we will define $\mathbf{G} \triangleq \mathbf{Y}\mathbf{K}\mathbf{Y}$.

**Multiple kernel learning.** *Multiple kernel learning* is simply the SVM problem with the additional complication that the kernel function is unknown, but is expressed as some function of other known kernel functions.

Following standard practice [18] we assume that the kernel function is a *convex combination* of other kernel functions; i.e., that there is some set of coefficients $\mu_i > 0$, that $\sum\mu_i = 1$, and that $\kappa = \sum\mu_i\kappa_i$ (which implies that the Gram matrix version is $\mathbf{K} = \sum\mu_i\mathbf{K}_i$). We regularize by setting $\text{tr}(\mathbf{K}) = 1$ [18]. The dual problem then takes the following form [18]:

$$\max_{\mathbf{K}}\min_{\alpha} \quad \frac{1}{2}\alpha^{\top}\mathbf{G}\alpha - \alpha^{\top}\mathbf{1} \tag{3.3}$$
$$\text{s.t.} \quad \mathbf{K} = \sum_{i=1}^{m}\mu_i\mathbf{K}_i, \quad \text{tr}(\mathbf{K}) = 1, \quad \mathbf{K} \succeq 0, \quad \mu \geq 0$$

When juxtaposed with (3.1) and (3.2), this can be interpreted as searching for the kernel that maximizes the shortest (kernel) distance between polytopes.

## 4 Our Algorithm

The MKL formulation of (3.3) can be transformed (as we shall see later) into a quadratically-constrained quadratic problem that can be solved by a number of different solvers [18, 1, 27]. However, this approach requires a memory footprint of $\Theta(mn^2)$ to store all kernel matrices. Another approach would be to exploit the min-max structure

of (3.3) via an alternating optimization: note that the problem of finding the shortest distance between polytopes for a fixed kernel is merely the standard SVM problem. There are two problems with this approach: (a) standard SVM algorithms do not scale well with $m,n$, and (b) it is not obvious how to adjust kernel weights in each iteration.

**Overview.** Our solution exploits the fact that a QCQP is a special case of a general SDP. We do this in order to apply the *combinatorial* primal-dual matrix multiplicative weight update (MMWU) algorithm of Arora & Kale [3]. While the generic MMWU has expensive steps (a linear program and matrix exponentiation), we show how to exploit the structure of the MKL QCQP to yield a very simple alternating approach. In the "forward" step, rather than solving an SVM, we merely find two support vector that are "most violating" normal to the current candidate hyperplane (in the lifted feature space). In the "backward" step, we reweight the kernels involved using a matrix exponentiation that we reduce to a *closed form* computation without requiring expensive matrix decompositions. Our speedup comes from the facts that (a) the updates to support vectors are sparse (at most two in each step) and (b) that the backward step can be computed very efficiently. This allows us to reduce our memory footprint to $O(mn)$.

**QCQPs and SDPs.** We start by using an observation due to Lanckriet et al. [18] to convert (3.3)[1] into the following QCQP:

$$\max_{\alpha,s} \quad (2\alpha^{\top}\mathbf{1} - s) \tag{4.1}$$
$$\text{s.t.} \quad s \geq \frac{1}{r_i}\alpha^{\top}\mathbf{G}_i\alpha, \quad \alpha^{\top}\mathbf{y} = 0, \quad \alpha \geq 0$$

where $\mathbf{G}_i = \mathbf{Y}\mathbf{K}_i\mathbf{Y}$, $r \in \mathbb{R}^m$, and $r_i = \text{tr}(\mathbf{K}_i)$.

Next, we rewrite (4.1) in canonical SDP form in order to apply the MMWU framework:

$$\omega^* = \max_{\alpha,s} \quad 2\alpha^{\top}\mathbf{1} - s \tag{4.2}$$
$$\text{s.t.} \quad \forall i \in [1..m] \quad \mathbf{Q}_i(\alpha) = \begin{pmatrix} \mathbf{I}_n & \mathbf{A}_i\alpha \\ (\mathbf{A}_i\alpha)^{\top} & s \end{pmatrix},$$
$$\mathbf{Q}_i(\alpha) \succeq \mathbf{0}, \quad \alpha^{\top}\mathbf{y} = 0, \quad \alpha \geq \mathbf{0}.$$

where $\mathbf{A}_i^{\top}\mathbf{A}_i = \frac{1}{r_i}\mathbf{G}_i$ for all $i \in [0..m]$.

**The MMWU framework.** We give a brief overview of the MMWU framework of Arora & Kale [3] (for more details, the reader is directed to Satyen Kale's thesis [14]).

---

[1]We note that (4.1) is the *hard-margin* version of the MKL problem. The standard soft-margin variants can also be placed in this general framework [18]. For the 1-norm soft margin, we add the constraint that all terms of $\alpha$ are upper bounded by the margin constant $C$. For the 2-norm soft margin, another term $\frac{1}{C}\alpha^{\top}\alpha$ appears in the objective, or we can simply add a constant multiple of $\mathbf{I}$ to each $\mathbf{G}_i$.

The approach starts with a "guess" $\omega$ for the optimal value $\omega^*$ of the SDP (and uses a binary search to find this guess interleaved with runs of the algorithm). Assuming that this guess at the optimal value is correct, the algorithm then attempts to find either a feasible primal (**P**) or dual assignment such that this guess is achieved.

---

**Algorithm 1** MMWU template [3]

---

**Input:** $\varepsilon$, primal $\mathbf{P}^{(1)}$, rounds $T$, guess $\omega$
  **for** $t = 1 \ldots T$ **do**
    *forward:* Compute update to $\alpha^{(t)}$ based on constraints, $\mathbf{P}^{(t)}$
    and $\alpha^{(t)}$
    *backward:* Compute $\mathbf{M}^{(t)}$ from constraints and $\alpha^{(t)}$.
$$\mathbf{W}^{(t+1)} \leftarrow e^{-\varepsilon \sum_{t=1}^{t} \mathbf{M}^{(t)}}$$
$$\mathbf{P}^{(t+1)} \leftarrow \frac{\mathbf{W}^{(t+1)}}{\mathrm{Tr}(\mathbf{W}^{(t+1)})}$$
  **end for**
**Output:** $\mathbf{P}^{(T)}$

---

The process starts with some assignment to $\mathbf{P}^{(1)}$ (typically the identity matrix $\mathbf{I}$). If this assignment is both primal feasible and at most $\omega$, the process ends. Else, there must be some assignment to $\alpha$ (the dual) that "witnesses" this lack of feasibility or optimality, and it can be found by solving a linear program using the current primal/dual assignments and constraints (i.e., is positive, has dual value at least $\omega$, and satisfies constraints (4.1)).

The primal constraints and $\alpha$ are then used to guide the search for a new primal assignment $\mathbf{P}^{(t+1)}$. They are combined to form the matrix $\mathbf{Q}_i(\alpha^{(t)})$ (see (4.1)), and then adjusted to form an "event matrix" $\mathbf{M}^{(t)}$ (see Paragraph "the backward step" for details)[2]. Exponentiating the sum of all the observed $\mathbf{M}^{(t)}$ so far, the algorithm exponentially re-weights primal constraints that are more important, and the process repeats. By minimizing the loss, the assignments to $\mathbf{P}^{(t)}$ and $\alpha^{(t)}$ are guaranteed to result in an SDP value that approximates $\omega^*$ within a factor of $(1 + \varepsilon)$.

**Our algorithm.** We now adapt the above framework to solve the MKL SDP given by (4.2). As we will explain below, we can assign $\omega^*$ *a priori* in most cases and we can solve our problem with only one round of feasibility search. We denote the dual update in iteration $t$ by $\alpha^{(t)}$, the $i$th event matrix in iteration $t$ by $\mathbf{M}_i^{(t)}$ and the $i$th primal variable (matrix) in iteration $t$ by $\mathbf{P}_i^{(t)}$. $\mathbf{P}_i^{(t)}$ is closely related to the desired primal kernel coefficients $\mu_i$. We denote $\alpha = \sum_i \alpha^{(i)}$ as the accumulated dual assignment thus far and $\mathbf{M}_i = \sum_t \mathbf{M}_i^{(t)}$ as the accumulated $i$th event matrix.

**The backward step.** It will be convenient to explain the backward step first. Given $\alpha^{(t)}$ and $\mathbf{Q}_i(\alpha^{(t)})$, we define

---

[2]$\mathbf{M}^{(t)}$ generalizes the loss incurred by experts in traditional MWU – by deriving $\mathbf{M}^{(t)}$ from the SDP constraints, the duality gap of the SDP takes the role of the loss.

$\mathbf{M}_i^{(t)} \triangleq \frac{1}{2\rho}(\mathbf{Q}_i(\alpha^{(t)}) + \rho \mathbf{I}_{n+1})$ where $\rho$ is a rate parameter to be set later. Note that $\mathbf{M}_i^{(t)}$ (and $\mathbf{M}^{(t)}$) is "almost-diagonal", taking the form $\begin{bmatrix} a\mathbf{I}_n & \mathbf{u} \\ \mathbf{u}^\top & a \end{bmatrix}$. Such matrices can be exponentiated in closed form.

**Lemma 4.1.** *The exponential of a matrix in the form* $\begin{pmatrix} a\mathbf{I}_n & \mathbf{u} \\ \mathbf{u}^\top & a \end{pmatrix}$, *where* $a \geq 0$ *and* $\hat{\mathbf{u}} = \mathbf{u}/\|\mathbf{u}\|$, *is*

$$e^a \left[ \begin{pmatrix} \cosh\|\mathbf{u}\|\hat{\mathbf{u}}\hat{\mathbf{u}}^\top & \sinh\|\mathbf{u}\|\hat{\mathbf{u}} \\ \sinh\|\mathbf{u}\|\hat{\mathbf{u}}^\top & \cosh\|\mathbf{u}\| \end{pmatrix} + \begin{pmatrix} \mathbf{I}_n - \hat{\mathbf{u}}\hat{\mathbf{u}}^\top & 0 \\ 0 & 0 \end{pmatrix} \right].$$

Lemma 4.1 (proved in Appendix A.1) implies that we can exponentiate the event matrix $\mathbf{M}^{(t)}$ (see Algorithm 1) quickly, as promised. In particular, we set $\mathbf{P}_i^{(t+1)} = c \exp(-\varepsilon \mathbf{M}_i)$ where $c$ normalizes the matrix to have unit trace[3]. Note that the structure of $\mathbf{P}^{(t)}$ also allows us to avoid storing it explicitly, since $(a\mathbf{I}) \bullet (b\hat{\mathbf{u}}\hat{\mathbf{u}}^\top) = ab$. We need only store the coefficients of the blocks of the $\mathbf{P}_i^{(t)}$.

**The forward step.** In the forward step, we wish to check if our primal solution **P** is feasible and optimal, and if not find updates to $\alpha^{(t)}$. In order to do so, we apply the MMWU template. Omitting the iteration parameter $t$ for clarity, the goal now is to find $\alpha$ such that

$$\sum_i \mathbf{Q}_i(\alpha) \bullet \mathbf{P}_i \geq 0, \; \alpha \geq 0, \; \alpha^\top \mathbf{y} = 0, \text{ and } \alpha^\top \mathbf{1} = 1.$$

The existence of such a $\alpha$ will prove that the current guess $\mathbf{P}^{(t)}$ is either primal infeasible or suboptimal (see Arora & Kale [3] for details).

We now exploit the structure of $\mathbf{P}^{(t)}$ given by Lemma 4.1. In particular, let $p_i^{11} = p_i^{22} = e^a \cosh\|\mathbf{u}\|, p_i^{12} = -e^a \sinh\|\mathbf{u}\|$. Then the first inequality above distills to

$$\left( \sum_i 2p_i^{12}\mathbf{g}_i \right)^\top \alpha^{(t)} \geq -1, \qquad (4.3)$$

where $\mathbf{g}_i = (\frac{1}{r_i}\mathbf{G}_i\alpha)/(\frac{1}{r_i}\alpha^\top\mathbf{G}_i\alpha)^{1/2}$. If we let $\mathbf{g} = \sum_i 2p_i^{12}\mathbf{g}_i$ (which can be calculated at the end of the *backward* step), then we have simply $\mathbf{g}^\top\alpha \geq -1$ which is a simple collection of linear constraints that can *always* be satisfied[4].

---

[3]Since Lemma 4.1 provides a closed-form solution to matrix exponentiation in this limited case, we need only worry about whether the individual functions that make up the elements of the exponentiation are numerically stable. Since $\mathbf{P}_i^{(t+1)}$ is normalized to have unit trace, however, we can add any additive term to $a$ to adjust the input of $e^a$ to a range that won't overflow. Moreover, $\cosh(x)$ and $\sinh(x)$ converge exponentially to $\exp(x)/2$, so if $\|\mathbf{u}\|$ is large, we can adjust $a$ to prevent instability in the first matrix term.

[4]The current margin borders a convex combination of points from each side. If we could not find a point such that the inequality is satisfied, then no point from the convex combination can be found on or past the margin, which is impossible.

Geometrically, $\mathbf{g}$ gives us a way to examine the training points that are farthest away from the margin. The higher a value $g_j$ is, the more it violates the current decision boundary. In order to find a $\alpha$ that satisfies (4.3), we simply choose the highest elements of $\mathbf{g}$ that correspond to both positive and negative labels, then set each corresponding entry in $\alpha$ to $1/2$. Algorithm 2 describes the pseudo-code for this process.

---

**Algorithm 2** FIND-$\alpha$

---

**Input:** $\mathbf{y}$, $\mathbf{g}$
$\quad P \leftarrow \{i \mid \mathbf{y}_i = 1\}, N \leftarrow \{i \mid \mathbf{y}_i = -1\}$
$\quad i_P \leftarrow \arg\max_{i \in P} \mathbf{g}_i, \ i_N \leftarrow \arg\max_{i \in N} \mathbf{g}_i$
$\quad \alpha \leftarrow \mathbf{0}$
$\quad \alpha_{i_P} \leftarrow 1/2, \ \alpha_{i_N} \leftarrow 1/2$
$\quad$ **return** $\alpha$
**Output:** $\alpha$ s.t. $\alpha \geq 0, \ \alpha^\top \mathbf{1} = 1, \ \alpha^\top \mathbf{y} = 0$

---

**Avoiding binary search for $\omega$.** The objective function in (4.2) is *linear*, so we can scale $s$ and $\alpha$ and use the fact that $s = \alpha^\top \mathbf{1} = \omega$ to transform the problem[5]:

$$\text{find} \quad \alpha \quad \text{s.t.}$$
$$1/\omega \geq \frac{1}{r_i}\alpha^\top \mathbf{G}_i\alpha, \quad \alpha^\top \mathbf{y} = 0, \quad \alpha^\top \mathbf{1} = 1, \quad \alpha \geq 0,$$

where $\alpha = \omega\alpha$. The first constraint can be transformed back into an optimization; that is, $\min_\omega \max_{\alpha,i} \frac{1}{r_i}\alpha^\top \mathbf{G}_i\alpha$, subject to the remaining linear constraints. Because $\omega$ does not figure into the maximization, we can compute $\omega$ simply by maximizing $\frac{1}{r_i}\alpha^\top \mathbf{G}_i\alpha$. Practically, this means that we simply add the constraint $\alpha^\top \mathbf{1} = 1$, and the "guess" for $\omega$ is set to 1. We then *know* the objective, and only one iteration is needed, so the binary search is eliminated.

### 4.1 Extracting the solution from the MMWU

We start by observing that $\sum_{i=1}^m \mathbf{Q}_i \bullet \mathbf{P}_i = 0$ (by complementary slackness), which can rewritten as

$$\sum_{i=1}^m \frac{2p_i^{!2}}{r_i}\left(\frac{r_i}{\alpha^\top \mathbf{G}_i\alpha}\right)^{1/2}\alpha^\top \mathbf{G}_i\alpha = 1. \quad (4.4)$$

Now recall (from section 3) that $\alpha^\top \mathbf{G}\alpha = \sum_{i=1}^m \mu_i \cdot \alpha^\top \mathbf{G}_i\alpha$, and we also use the fact that $\alpha^\top \mathbf{G}\alpha = \alpha^\top \mathbf{1} = \omega = 1$. Combining the above two we have:

$$\sum_{i=1}^m \mu_i \cdot \alpha^\top \mathbf{G}_i\alpha = 1 \quad (4.5)$$

---

[5]This fact follows from the KKT conditions for the original problem. The support constraints of the SVM problem can be written as $\mathbf{G}\alpha + b\mathbf{y} \geq \mathbf{1}$. If we multiply both sides of this inequality by $\alpha^\top$ then it becomes an equality (by complementary slackness): $\alpha^\top \mathbf{G}\alpha = \alpha^\top \mathbf{1}$. $s$ is a substitution for $\alpha^\top \mathbf{G}\alpha$ in the MKL problem [18] so $s = \alpha^\top \mathbf{1} = \omega$ as well.

Matching (4.4) with (4.5) suggests that $\frac{2p_i^{!2}}{r_i}\left(\frac{r_i}{\alpha^\top \mathbf{G}_i\alpha}\right)^{1/2}$ is the appropriate choice for $\mu_i$.

### 4.2 Putting it all together

Algorithm 3 summarizes the discussion in this section. The parameter $\varepsilon$ is the error in approximating the objective function, but its connection to classification accuracy is loose. We set the actual value of $\varepsilon$ via cross-validation (see Section 5). The parameter $\rho$ is the *width* of the SDP, a parameter that indicates how much the solution can vary at each step. $\rho$ is equal to the maximum absolute value of the eigenvalues of $\mathbf{Q}_i(\alpha^{(t)})$, for any $i$ [3]. We show that $\rho^2 \leq 3/2$ in Appendix A.3.

**Running time.** Every iteration of Algorithm 3 will require a call to FIND-$\alpha$, a call to EXPONENTIATE-$M$ and an update to $\mathbf{G}_i\alpha$ and $\alpha^\top \mathbf{G}_i\alpha$. FIND-$\alpha$ requires a linear search for two maxima in $\mathbf{g}$, so the first is $O(n)$. The latter are each $O(mn)$, which dominate FIND-$\alpha$.

Algorithm 3 requires a total of $T$ iterations at most, where $T = \frac{8\rho^2}{\varepsilon^2}\ln(n)$. Since we only require one run of the main algorithm, the running time is bounded by $O\left(mn\ln(n)\frac{1}{\varepsilon^2}\right)$.

---

**Algorithm 3** MWUMKL

---

**Input:** $\mathbf{g}^{(1)} = \mathbf{0}$;
$\quad \rho$, the width;
$\quad \varepsilon$, the desired approximation error
$\quad$ Set $\varepsilon' = -\ln(1 - \frac{\varepsilon}{2\rho})$
$\quad$ Set $T = \frac{8\rho^2}{\varepsilon^2}\ln(n)$
$\quad$ **repeat** $\{T \text{ times}\}$
$\quad\quad$ Get $\alpha^{(t)}$ from Algorithm 2
$\quad\quad$ **if** Algorithm 2 failed **then**
$\quad\quad\quad$ Return
$\quad\quad$ **end if**
$\quad\quad$ Update $\alpha = \alpha + \alpha^{(t)}$
$\quad\quad$ Set $\mathbf{M}_i^{(t)} = \frac{1}{2\rho}\left(\mathbf{Q}_i(\alpha^{(t)}) + \rho\mathbf{I}_{n+1}\right)$
$\quad\quad$ Set $\mathbf{W}_i^{(t)} = e^{-\varepsilon'\sum_{t=1}^T \mathbf{M}_i^{(t)}}$
$\quad\quad$ Set $\mathbf{P}_i^{(t+1)} = \mathbf{W}_i^{(t)}/\text{tr}(\mathbf{W}_i^{(t)})$
$\quad\quad$ Compute $\mathbf{g}^{(t+1)}$ from $\mathbf{P}^{(t+1)}$, $\{\mathbf{G}_i\}$, and $\alpha$
$\quad$ **until** $t = T$
$\quad$ Return $\frac{1}{T}\alpha$, $\mathbf{P}^{(T+1)}$

---

## 5 Experiments

In this section we compare the empirical performance of MWUMKL with other multiple kernel learning algorithms. Our results have two components: (a) *qualitative* results that compares test accuracies on small scale datasets, and (b) *scalability* results that compares training time on larger datasets.

We compare MWUMKL with the following baselines:

(a) UNIFORM (uniformly weighted combination of kernels), and (b) LibLinear [11] with Nyström kernel approximations for each kernel (hereafter referred to as LIBLINEAR+). We evaluate these MKL methods on binary datasets from UCI data repository. They include: (a) small datasets *Iono*, *Breast Cancer*, *Pima*, *Sonar*, *Heart*, *Vote*, *WDBC*, *WPBC*, (b) medium dataset *Mushroom*, and (c) comparatively larger datasets *Adult*, *CodRna*, and *Web*.

Classification accuracy and kernel scalability results are presented on small and medium datasets (with many kernels). Scalability results (with 12 kernels due to memory constraints) are provided for large datasets. Finally, we show results for lots of kernels on small data subsets.

**Uniform kernel weights.** UNIFORM is simply LibSVM [6] run with a kernel weighted equally amongst all of the input kernels (where the kernel weights are normalized by the trace of their respective Gram matrices first). The performance of UNIFORM is on par or better than LIBLINEAR+ on many datasets (see Figure 2) and the time is similar to MWUMKL. However UNIFORM does not scale well due to the poor scaling of LibSVM beyond a few thousand samples (see Figure 3), because of the need to hold the entire Gram matrix in memory[6]. We employ Scikit-learn [23] because it offers efficient access to LibSVM.

**LibLinear [11] with Nyström kernel approximations [31, 34] (LIBLINEAR+).** One important observation about multiple kernel learning is that UNIFORM performs as well or better than many MKL algorithms with better efficiency. Along this same line of thought, we should consider comparison against methods that are as simple as possible. One of the very simplest algorithms to consider is to use a linear classifier (in this case, LibLinear [11]), and transform the features of the data with a kernel approximation. For our purposes, we use Nyström approximations as described by Williams & Seeger [31] and discussed further by Yang et al. [34]. Because LibLinear is a primal method, we don't need to scale each kernel – each kernel manifests as a set of features, which the algorithm weights by definition.

For the Nyström feature transformations, one only needs to specify the kernel function and the number of sample points desired from the data set. We usually use 150 points, unless memory constraints force us to use fewer. Theoretically, if $s$ is the number of sample points, $n$ the number of data points, and $m$ the number of kernels, then we would need space to store $O(snm)$ double-precision floats. With regard to time, the training task is very rapid – the transformation

is the bottleneck (requiring $O(s^2mn)$ time to transform every point with every kernel approximation).

We employ Scikit-learn [23] for implementations of both the linear classifier and the kernel approximation because (a) this package offloads linear support-vector classification to the natively-coded LibLinear implementation, (b) it offers a fast kernel transformation using the NumPy package, and (c) Scikit-learn makes it very easy and efficient to chain these two implementations together. In practice this method is very good and very fast for low numbers of kernels (see Figures 2, 4a, and 4b). For high numbers of kernels, this scaling breaks down due to time and memory constraints (see Figure 5).

**Legacy MKL implementations.** In all cases, we omit the results for older MKL algorithm implementations such as (a) SILP [26], (b) SDPMKL [18], (c) SIMPLEMKL [25], (d) LEVELMKL [32], and (e) GROUPMKL [33] which take significantly longer to complete, have no significant gain in accuracy, and do not scale to any datasets larger than a few thousand samples. For example, on *Sonar* (one of the smallest sets in our pool), each iteration of SILP takes about 4500 seconds on average whereas UNIFORM requires 0.03 seconds on average.

| Size | Dataset | #Points | #Dim |
|------|---------|---------|------|
| | *Breast Cancer* | 683 | 9 |
| | *Heart* | 270 | 13 |
| | *Iono* | 351 | 33 |
| Small | *Pima* | 768 | 8 |
| | *Sonar* | 208 | 60 |
| | *Vote* | 435 | 16 |
| | *WDBC* | 569 | 30 |
| | *WPBC* | 198 | 33 |
| Medium | *Mushroom* | 8124 | 112 |
| | *Adult* | 39073 | 123 |
| Large | *CodRna* | 47628 | 8 |
| | *Web* | 64700 | 300 |

Table 1: Datasets used in experiments.

**Experimental parameters.** Similar to Rakotomamonjy et al. [25] and Xu et al. [33], we test our algorithms on a base kernel family of 3 polynomial kernels (of degree 1 to 3) and 9 Gaussian kernels. Contrary to [25, 33], however, we test with Gaussian kernels that have a tighter range of bandwidths ($\{2^0, 2^{1/2}, \ldots, 2^4\}$, instead of $\{2^{-3}, 2^{-2}, \ldots, 2^5\}$). The reason for this last choice is that our method actively seeks solutions for each of the kernels, and kernels that encourage overfitting the training set (such as low-bandwidth Gaussian kernels) pull MWUMKL away from a robust solution.

For small datasets, kernels are constructed using each single feature and are repeated 30 times with different train/test partitions. For medium and large datasets, due to memory constraints on LIBLINEAR+, we test only on

---

[6]This is true even when LibSVM is told to use one kernel, which it can compute on the fly – the scaling of LibSVM is $O(n^2)$ - $O(n^3)$ [6], poor compared to MWUMKL and LIBLINEAR+ with increasing sample size.

12 kernels constructed using all features, and repeat only 5 times. All kernels are normalized to trace 1. Results from small datasets are presented with a 95% confidence interval that the median lies in the range. Results from medium-large datasets present the median, with the min and max values as a range around the median. In each iteration, 80% of the examples are randomly selected as the training data and the remaining 20% are used as test data. Feature values of all datasets have been scaled to $[0, 1]$. SVM regularization parameter $C$ is chosen by cross-validation. For example, in Figure 2 results are presented for the best value of $C$ for each dataset and algorithm.

For MWUMKL, we choose $\varepsilon$ by cross-validation. Most datasets get $\varepsilon = 0.2$, but the exceptions are *Web* ($\varepsilon = 0.07$), *CodRna* ($\varepsilon = 0.07$), and *Adult* ($\varepsilon = 0.05$). Contrary to existing works we do not compare the number of SVM calls (as MWUMKL does not explicitly use an underlying SVM) and the number of kernels selected.

Experiments were performed on a machine with an Intel® Core™ 2 Quad CPU (2.40 GHz) and 2GB RAM. All methods have an outer test harness written in Python. MWUMKL also uses a test harness in Python with an inner core written in C++.



Figure 3: *CodRna* ($n = 59535$, $d = 8$) with 12 kernels.

for the remaining experiments on large datasets, we compare MWUMKL with LIBLINEAR+. In Figures 4a and 4b, we choose a random partition of train and test, and then train with increasing proportions of the training partition (but always test with the whole test partition). With more data, our algorithm settles in to be competitive with LIBLINEAR+.
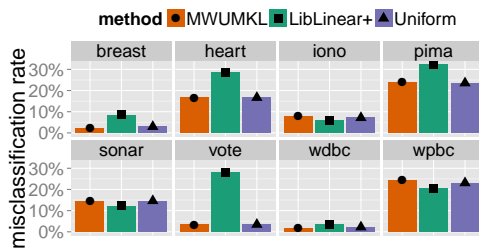


Figure 2: Median misclassification rate for small datasets.

**Accuracy.** On small datasets our goal is to show that MWUMKL compares favorably with LIBLINEAR+ and UNIFORM in terms of test accuracies.

In Figure 2 we present the median misclassification rate for each small dataset over 30 random training/test partitions. In each case, we train the classifier with 12 kernels for each feature in the dataset, and each kernel only operates on one feature. We are able either to beat the other methods or remain competitive with them.

**Data Scalability.** Both MWUMKL and LIBLINEAR+ are much faster as compared with UNIFORM. At this point, *Adult*, *CodRna*, and *Web* are large enough datasets that UNIFORM fails to complete because of memory constraints. This can be seen in Figure 3, where we plot training time versus the proportion of the training data used – the training time taken by UNIFORM rises sharply and we are unable to train on this dataset past 11907 points. Hence,
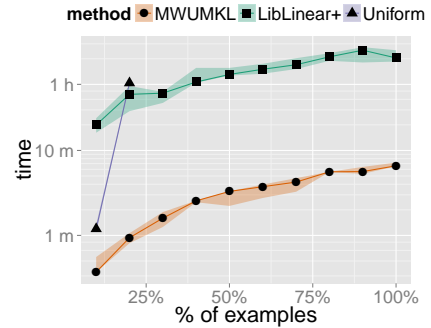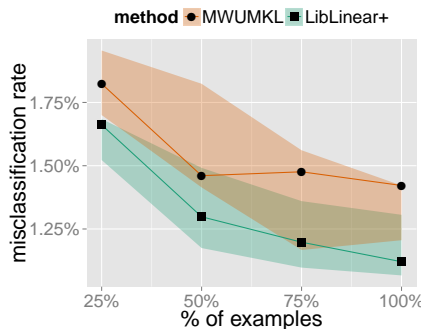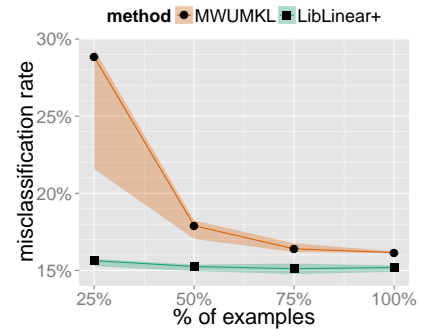




Figure 4: *Adult* ($n = 48842$, $d = 123$) and *Web* ($n = 64700$, $d = 300$) with $m = 12$ kernels

**Kernel Scalability.** We aim to demonstrate not only that MWUMKL performs well with the number of examples, but also that it performs well against the number of kernels. In fact, for an MKL algorithm to be truly scalable it should do well against *both* examples and kernels.

For kernel scalability, we present the training times for the best parameters of several of the datasets, divided by the number of kernels used, versus the size of the dataset (see Figure 5). We divide time by number of kernels because time scales very close to linearly with the number of kernels for all methods. Also presented are log-log models fit to the data, and the median of each experiment is plotted as a point.

We report the time for the same experiments that produced the results in Figure 2, and also train on increasing proportions of *Mushroom* (1625, 3250, 4875, and 6500 examples) with 1344 per-feature kernels. With these selections, we are testing $mn$ in the neighborhood of 8.7 million elements.
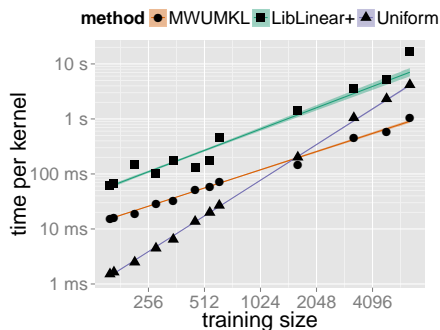


Figure 5: Time per kernel vs. data size for small and medium data sets (log-log).

As expected, UNIFORM scales quadratically or more with the number of examples, performing very well at the lower range. The number of examples from *Mushroom* is not so high that LibSVM runs out of memory, but we do see the algorithm's typical scaling.

LIBLINEAR+ shows slightly superlinear scaling, with a high multiplier due to the matrix computations required for the feature transformations. As we run the algorithm on *Mushroom*, the number of samples taken for the kernel approximations is reduced so that the features can fit in machine memory. Even so, this reduction doesn't offer any help to the scaling and at 6500 examples with 1344 kernels, training time is several hours.

Even though we reduced the number of samples for LIBLINEAR+, MWUMKL outperforms both UNIFORM and LIBLINEAR+ when both examples and kernels are greater than about $10^3$.

**Dynamic Kernels.** We also present results for a few datasets with lots of kernels. By computing columns of the kernel matrices on demand, we can run with a memory footprint of $O(mn)$, improving scalability without affecting solution quality (a technique also used in SMOMKL [30]). Table 2 shows that we can indeed scale well beyond tens of thousands of points, as well as many kernels.

| Dataset | #Points | #Kernels | Time |
|---------|---------|----------|------|
| *Adult* | 39073 | 3 | 13 minutes |
| *CodRna* | 47628 | 3 | 147 seconds |
| *Sonar* 1M | 208 | 1000000 | 3.65 hours |

Table 2: MWUMKL with on-the-fly kernel computations.

We choose the above datasets to compare against another work on scalable MKL [13]. Jain et al. [13] indicate the ability to deal with millions of kernels, but in effect the technique also has a memory footprint of $\Omega(mn)$ (the footprint of MWUMKL is $\Theta(mn)$, in contrast). This limits any such approach to *either* many kernels *or* many points, but not both.

Since the work in Jain et al. [13] does not provide accuracy numbers, a direct head-to-head comparison is difficult to make, but we can make a subjective comparison. The above table shows times for MWUMKL with accuracy similar to or better than what LIBLINEAR+ can achieve on the same datasets. The time numbers we achieve are similar in order of magnitude when scaled to the number of kernels demonstrated in Jain et al. [13].

# 6 Conclusions and Future Work

We have presented a simple, fast and easy to implement algorithm for multiple kernel learning. Our proposed algorithm develops a geometric reinterpretation of kernel learning and leverages fast MMWU-based routines to yield an efficient learning algorithm. Detailed empirical results on data scalability, kernel scalability and with dynamic kernels demonstrate that we are significantly faster than existing legacy MKL implementations and outpeform LIBLINEAR+ as well as UNIFORM.

Our current results are for a single machine. As mentioned earlier, one of our future goals is to add parallellization techniques to improve the scalability of MWUMKL over data sets that are large and use a large number of kernels. The MWUMKL algorithm lends itself easily to the *bulk synchronous parallel* (BSP) framework [28], as most of the work is done in the loop that updates $\mathbf{G}\alpha$ (see the last line of the loop in Algorithm 3). This task can be "sharded" for either kernels or data points, and scalability of $O(mn)$ would not suffer under BSP. Since there are many BSP frameworks and tools in use today, this is a natural direction to experiment.

# 7 Acknowledgments

# References

[1] Andersen, E. D. and Andersen, K. D. *The MOSEK interior point optimization for linear programming: an implementation of the homogeneous algorithm*, pp. 197–232. Kluwer Academic Publishers, 1999.

[2] Argyriou, Andreas, Hauser, Raphael, Micchelli, Charles A., and Pontil, Massimiliano. A DC-programming algorithm for kernel selection. In *ICML*, Pennsylvania, USA, 2006.

[3] Arora, Sanjeev and Kale, Satyen. A combinatorial, primal-dual approach to semidefinite programs. In *STOC*, pp. 227–236, New York, NY, USA, 2007. ACM.

[4] Bach, Francis R., Lanckriet, Gert R. G., and Jordan, Michael I. Multiple kernel learning, conic duality, and the SMO algorithm. In *ICML*, pp. 6–, New York, NY, USA, 2004. ACM.

[5] Bennett, Kristin P. and Bredensteiner, Erin J. Duality and geometry in SVM classifiers. In *ICML*, pp. 57–64, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[6] Chang, Chih-Chung and Lin, Chih-Jen. LIBSVM: A library for support vector machines. *ACM TIST*, 2(3):27:1–27:27, May 2011.

[7] Cortes, Corinna. Invited talk: Can learning kernels help performance? In *ICML*, Montreal, Canada, 2009.

[8] Cortes, Corinna, Mohri, Mehryar, and Rostamizadeh, Afshin. Learning non-linear combinations of kernels. In *NIPS*, pp. 396–404, Vancouver, Canada, 2009.

[9] Cortes, Corinna, Mohri, Mehryar, and Rostamizadeh, Afshin. Two-stage learning kernel algorithms. In *ICML*, pp. 239–246, Haifa, Israel, 2010.

[10] Cristianini, Nello, Shawe-Taylor, John, Elisseeff, André, and Kandola, Jaz S. On kernel-target alignment. In *Innovations in Machine Learning*, pp. 205–256. Springer, 2006.

[11] Fan, Rong-En, Chang, Kai-Wei, Hsieh, Cho-Jui, Wang, Xiang-Rui, and Lin, Chih-Jen. LIBLINEAR: A library for large linear classification. *JMLR*, 9:1871–1874, 2008.

[12] Gretton, Arthur, Borgwardt, Karsten M, Rasch, Malte, Schölkopf, Bernhard, and Smola, Alexander J. A kernel method for the two-sample problem. In *NIPS*, pp. 513–. MIT, 2007.

[13] Jain, Ashesh, Vishwanathan, S.V.N., and Varma, Manik. SPF-GMKL: Generalized multiple kernel learning with a million kernels. In *KDD*, pp. 750–758, New York, NY, USA, 2012. ACM.

[14] Kale, Satyen. *Efficient algorithms using the multiplicative weights update method*. PhD thesis, Princeton University, 2007.

[15] Kloft, Marius, Brefeld, Ulf, Sonnenburg, Sören, Laskov, Pavel, Müller, Klaus-Robert, and Zien, Alexander. Efficient and accurate Lp-norm multiple kernel learning. In *NIPS*, pp. 997–1005, Vancouver, Canada, 2009.

[16] Kloft, Marius, Brefeld, Ulf, Sonnenburg, Sören, and Zien, Alexander. $l_p$-norm multiple kernel learning. *JMLR*, 12: 953–997, 2011.

[17] Kumar, Abhishek, Niculescu-Mizil, Alexandru, Kavukcuoglu, Koray, and Daume, Hal III. A binary classification framework for two stage multiple kernel learning. In *ICML*, pp. 1295–1302, 2012.

[18] Lanckriet, Gert R. G., Cristianini, Nello, Bartlett, Peter, Ghaoui, Laurent El, and Jordan, Michael I. Learning the kernel matrix with semidefinite programming. *JMLR*, 5:27–72, December 2004.

[19] Micchelli, Charles A. and Pontil, Massimiliano. Learning the kernel function via regularization. *JMLR*, 6:1099–1125, December 2005.

[20] Ong, Cheng Soon, Smola, Alexander J., and Williamson, Robert C. Learning the kernel with hyperkernels. *JMLR*, 6: 1043–1071, 2005.

[21] Orabona, Francesco and Jie, Luo. Ultra-fast optimization algorithm for sparse multi kernel learning. In *ICML*, pp. 249–256, Bellevue, USA, 2011.

[22] Pavlidis, Paul, Weston, Jason, Cai, Jinsong, and Grundy, William Noble. Gene functional classification from heterogeneous data. In *Proc. Intl. Conf. on Computational Biology*, RECOMB '01, pp. 249–255, New York, NY, USA, 2001. ACM.

[23] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. Scikit-learn: Machine learning in Python. *JMLR*, 12:2825–2830, 2011.

[24] Phillips, Jeff M. and Venkatasubramanian, Suresh. A gentle introduction to the kernel distance. *CoRR*, abs/1103.1625, 2011.

[25] Rakotomamonjy, Alain, Bach, Francis, Canu, Stéphane, and Grandvalet, Yves. More efficiency in multiple kernel learning. In *ICML*, pp. 775–782, New York, NY, USA, 2007. ACM.

[26] Sonnenburg, Sören, Rätsch, Gunnar, Schäfer, Christin, and Schölkopf, Bernhard. Large scale multiple kernel learning. *JMLR*, 7:1531–1565, December 2006.

[27] Sturm, J. F. Using SeDuMi 1.02, a MATLAB toolbox for optimization over symmetric cones. *Optimization Methods and Software*, 11–12:625–653, 1999.

[28] Valiant, Leslie G. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990.

[29] Varma, Manik and Babu, Bodla Rakesh. More generality in efficient multiple kernel learning. In *ICML*, pp. 1065–1072, New York, NY, USA, 2009. ACM.

[30] Vishwanathan, S. V. N., Sun, Zhaonan, Ampornpunt, Nawanol, and Varma, Manik. Multiple kernel learning and the SMO algorithm. In *NIPS*, volume 22, pp. 2–, Vancouver, Canada, 2010.

[31] Williams, Christopher and Seeger, Matthias. Using the Nyström method to speed up kernel machines. In *NIPS*, pp. 682–688, 2001.

[32] Xu, Zenglin, Jin, Rong, King, Irwin, and Lyu, Michael R. An extended level method for efficient multiple kernel learning. In *NIPS*, pp. 1825–1832, Vancouver, Canada, 2008.

[33] Xu, Zenglin, Jin, Rong, Yang, Haiqin, King, Irwin, and Lyu, Michael R. Simple and efficient multiple kernel learning by group lasso. In *ICML*, pp. 1175–1182, Haifa, Israel, 2010.

[34] Yang, Tianbao, Li, Yu-Feng, Mahdavi, Mehrdad, Jin, Rong, and Zhou, Zhi-Hua. Nyström method vs random fourier features: A theoretical and empirical comparison. In *NIPS*, pp. 485–493, 2012.

[35] Ye, Jieping, Chen, Jianhui, and Ji, Shuiwang. Discriminant kernel and regularization parameter learning via semidefinite programming. In *ICML*, pp. 1095–1102, New York, NY, USA, 2007. ACM.

[36] Zien, Alexander and Ong, Cheng Soon. Multiclass multiple kernel learning. In *ICML*, pp. 1191–1198, New York, NY, USA, 2007. ACM.