KERNELS AND GEOMETRY OF MACHINE LEARNING

by

John Moeller

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computing

School of Computing

The University of Utah

May 2017

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of **John Moeller**

has been approved by the following supervisory committee members:

Suresh Venkatasubramanian,	Chair(s)	**26 Aug 2016** Date Approved
Hal Daumé III,	Member	**27 Sep 2016** Date Approved
P. Thomas Fletcher,	Member	**11 Sep 2016** Date Approved
Jeffrey Phillips,	Member	**29 Aug 2016** Date Approved
Vivek Srikumar,	Member	**5 Sep 2016** Date Approved

by **Ross Whitaker**, Chair/Dean of

the Department/College/School of **Computing**

and by **David B. Kieda**, Dean of The Graduate School.

ABSTRACT

The contributions in the area of kernelized learning techniques have expanded beyond a few basic kernel functions to general kernel functions that could be learned along with the rest of a statistical learning model. This dissertation aims to explore various directions in *kernel learning*, a setting where we can learn not only a model, but also glean information about the geometry of the data from which we learn, by learning a positive-definite (p.d.) kernel. Throughout, we can exploit several properties of kernels that relate to their *geometry* – a facet that is often overlooked.

We revisit some of the necessary mathematical background required to understand kernel learning in context, such as reproducing kernel Hilbert spaces (RKHSs), the reproducing property, the representer theorem, etc. We then cover kernelized learning with support vector machines (SVMs), multiple kernel learning (MKL), and localized kernel learning (LKL). We move on to Bochner's theorem, a tool vital to one of the kernel learning areas we explore.

The main portion of the thesis is divided into two parts: (1) kernel learning with SVMs, a.k.a. MKL, and (2) learning based on Bochner's theorem. In the first part, we present efficient, accurate, and scalable algorithms based on the SVM, one that exploits multiplicative weight updates (MWU), and another that exploits local geometry. In the second part, we use Bochner's theorem to incorporate a kernel into a neural network and discover that kernel learning in this fashion, continuous kernel learning (CKL), is superior even to MKL.

For my wife Aimee Núñez, who has loved and supported me through this whole disruptive journey. Aimee, I hold nothing but love for you in my heart, and I hope that I've earned your dedication and loyalty.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

NOTATION AND SYMBOLS

$\mathbb{Z}$	The integers
$[a..b]$	$\{i \in \mathbb{Z} \mid a \leq i \leq b,\ a, b \in \mathbb{Z}\}$
$\mathbb{R}$	The real numbers
$\mathbb{R}_+$	$\{x \in \mathbb{R} \mid x \geq 0\}$
$\mathbb{R}_{++}$	$\{x \in \mathbb{R} \mid x > 0\}$
$\mathbb{C}$	The complex numbers
$\mathsf{Re}(z)$	The real component of $z \in \mathbb{C}$
$\mathsf{Im}(z)$	The imaginary component of $z \in \mathbb{C}$
$\mathcal{A} \times \mathcal{B}$	Cartesian product of two sets $\mathcal{A}$ and $\mathcal{B}$
$\mathcal{A}^d$	Cartesian product of d many copies of $\mathcal{A}$
$\mathcal{F}^{m \times n}$	Set of $m \times n$ matrices of elements of field $\mathcal{F}$
$\mathbf{x}$	Vector x
$\hat{\mathbf{x}}$	Vector such that $\|\mathbf{x}\|\hat{\mathbf{x}} = \mathbf{x}$
$\mathbf{A}$	Matrix A
$\mathbf{0}_{m \times n}, \mathbf{0}_n, \mathbf{0}$	Zero vector or matrix
$\mathbf{1}_{m \times n}, \mathbf{1}_n, \mathbf{1}$	All-ones vector or matrix
$\mathbf{I}_n, \mathbf{I}$	$n \times n$ dentity matrix (just $\mathbf{I}$ if dimension is understood from context)
$\mathbf{A} \succeq 0$	$\mathbf{A}$ is positive semidefinite
$\mathbf{A} \succeq \mathbf{B}$	$\mathbf{A} - \mathbf{B}$ is positive semidefinite
$\text{diag}(\mathbf{a})$	The diagonal matrix $\mathbf{A}$ such that $A_{ii} = a_i$
$\mathbf{A}^\top$	The transpose of $\mathbf{A}$
$\text{tr}(\mathbf{A})$	The trace of $\mathbf{A}$ $(= \sum_i A_{ii} = \mathbf{I} \bullet \mathbf{A})$
$\mathbf{A} \circ \mathbf{B}$	Hadamard (elementwise) product of $\mathbf{A}$ and $\mathbf{B}$
	$(\mathbf{A} \circ \mathbf{B} = \mathbf{C}$ such that $C_{ij} = A_{ij}B_{ij})$
$\mathbf{A} \bullet \mathbf{B}$	$= \text{tr}(\mathbf{A}^\top\mathbf{B}) = \sum_{i,j} A_{ij}B_{ij} = \mathbf{B} \bullet \mathbf{A}$
$\|\cdot\|$	Norm
$\langle \cdot, \cdot \rangle$	Inner product
$\text{ReLU}(\cdot)$	Rectified linear function
	$(\text{ReLU}(x) = 0$ for $x < 0$, $\text{ReLU}(x) = x$ otherwise)

The softmax operator is a map $\mathbb{R}^d \to (0,1)^d$ that normalizes the input vector to the range $(0,1)$:

$$\text{softmax}(\mathbf{x}) = \left(\frac{\exp x_i}{\sum_{k=1}^{d} \exp x_k} \right)_{i=1}^{d}$$

ACKNOWLEDGEMENTS

First, I would like to recognize my advisor Suresh Venkatasubramanian for his tutelage these last several years. He has made me laugh about the frustrating parts, kicked me in the ass when I was slacking, talked me down when I was feeling defeated, and provided insight into how the process *really* works. Above all, he has been a *fair* mentor, backing me when I needed it and whipping me into shape when I needed it as well. Any PhD student should feel fortunate to have Suresh as a mentor.

I wouldn't be here without the rest of my committee: Hal, Tom, Jeff, and Vivek. You made this possible, so a big thank you from me to you. Thank you to the Scientific Computing Institute (SCI) for allowing us to use their resources for our experiments during their low utilization periods. Also thank you to the taxpayers of the USA, through the NSF, for providing me with a livable income and the means to get here. I'd also like to thank Satyen Kale for his PhD work and his excellent PhD dissertation, which clarified many aspects of my research.

Thank you to my labmates and classmen, particularly Samira, Protonu, Petey, and Mina, for laughs, cameraderie, and insight. A special thank you to Nathan Gilbert for getting me up to Suresh's lecture in 2008. Thanks to my academic friends Jake, Roni, Josh, and Steven for good conversation. Thanks to my nonacademic friends Mike, Kirsten, Jean, Robert, Dean, Dan, and Larry for keeping me grounded in reality. To my coauthors, Amir, Para, Avishek, Dustin, Sorelle, and Sarath, thank you for the collaboration and opportunities.

To my sister, Deborah Moeller, thank you for the good conversations, goofy jokes, and raising the level of discourse. To my mom, Marsha Moeller, thank you for feeding me, clothing me, protecting me, transporting me, and (sometimes successfully) anticipating my needs. To my dad, John Moeller, I wouldn't be earning a PhD were it not for your constant encouragement and support. Thank you for bringing home that monstrous Tektronix 4052 (I still remember its smell). It opened the door to a whole way of life for me.

CHAPTER 1

INTRODUCTION

Kernels were an important analytical tool before the field of machine learning was ever conceived. Their contribution to machine learning and other mathematical fields has been profound, opening the door to efficient and accurate computation of nonlinear models. Last decade, the contribution of kernels expanded beyond a few basic kernel functions to general kernel functions that could be learned along with the rest of a model. This direction continues to bear fruit. This dissertation aims to explore various directions in *kernel learning*, a setting where we can learn not only a model, but also glean information about the geometry of the data we learn, by learning a positive-definite (p.d.) kernel.

1.1 Thesis Statement

Kernels can be employed in a rich variety of ways by exploiting several of their mathematical properties. Additionally, because they are intrinsically *geometric* objects, we can build rigorous theory around them. Nevertheless, kernel methods suffer from a *scaling* problem. We can address this problem in multiple ways:

1. First, applying the right techniques can reduce this impact – we apply a well-regarded optimization technique and exploit the geometric structure of the problem.

2. Prediction is also a problem with kernel methods, because the cost of prediction scales linearly in the number of kernel representatives for the prediction function. *Localization* mitigates this problem – that is, by focusing a kernel's effect per-example, we can reduce the number of kernel representatives. We generalize local kernel learning methods and isolate the effect that the technique has on representer cardinality.

3. Finally, we can scale kernel learning to the largest datasets by exploiting neural network technology. Additionally, we show VC-dimension results on this type of

hypothesis class. By importing concepts such as sample complexity to (e.g.) deep learning, we can provide those areas with more tools to analyze them.

1.2 Organization of This Dissertation

Chapter 2 covers varied background material, including convex optimization, support vector machines (SVMs), kernels, reproducing kernel Hilbert spaces (RKHSs), multiple kernel learning (MKL), Bochner's theorem, and feedforward neural networks. This chapter will also cover prior work related to topics covered in this dissertation.

Chapter 3 presents a geometric formulation of the multiple kernel learning (MKL) problem. 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 datasets 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.

Chapter 4 describes localized kernel learning (LKL) and presents a unified framework to solve the problem. Most MKL methods seek the combined kernel that performs best over *every* training example, sacrificing performance in some areas to seek a global optimum. LKL overcomes this limitation by allowing the training algorithm to match a component kernel to the examples that can exploit it best. Several approaches to the LKL problem have been explored in the last several years. We unify many of these approaches under one simple system and describe an algorithm with improved performance. We also develop enhanced versions of existing algorithms, with an eye on scalability and performance.

Chapter 5 describes a new approach to kernel learning that establishes connections between the Fourier-analytic representation of kernels arising out of Bochner's theorem and a specific kind of feed-forward network using cosine activations. We analyze the complexity of this space of hypotheses and demonstrate empirically that our approach provides scalable kernel learning superior in quality to prior kernel learning approaches.

Chapter 6 discusses the contributions of this dissertation, and discusses future work directions.

CHAPTER 2

KERNEL LEARNING

2.1 Convex Optimization

Convex optimization is useful for machine learning since many machine learning problems can be expressed as a minimization of a convex function over a convex set.

Definition 1 (Boyd and Vandenberghe [16, sec. 4.2])**.** A convex optimization problem (or convex program) is one of the form

$$\begin{aligned} \min \quad & f_0(\mathbf{x}) \\ \text{s.t.} \quad & f_i(\mathbf{x}) \leq 0, \forall i \in [1..m] \\ & \mathbf{a}_j^\top \mathbf{x} = b_j, \forall j \in [1..p], \end{aligned}$$

where $\{f_0 \ldots f_m\}$ are convex functions.

Several categories of optimization problems fall under the "convex" category. The types that we will be concerned with fall into one of three categories: linear programs (LPs), quadratically constrained quadratic programs (QCQPs), and semidefinite programs (SDPs). We will mention others such as second order cone programs (SOCPs) where they are relevant. We will not go into detail describing the various varieties of convex programs, since they are described in fine detail in other material. We refer the reader to the very detailed book by Boyd and Vandenberghe [16, see chap. 4].

One topic vital to the content of this dissertation is *duality*. Duality is a property of optimization problems that says that every optimization problem, the *primal* problem, has a *dual* problem (technically a *Lagrange* dual) and that the optimum for one is a bound on the optimum for the other, and vice versa (a.k.a. *weak duality*). This property holds even when the problem is not convex. When the optima for the two are equal, *strong duality* holds. This usually means that the dual can be solved in lieu of the primal and that the primal variables can be recovered from the dual variables. Convex programs, with some basic criteria, have strong duals. We refer the reader again to Boyd and Vandenberghe [16, see chap. 5] for more details.

2.2 Support Vector Machines

Given a finite set of n example/label pairs belonging to $\mathcal{X} \times \mathcal{Y}$, where $\mathcal{Y} = \{-1, 1\}$, our task is to find a model $f(\cdot)$ that accurately predicts a new label y for some input $\mathbf{x}$. The support vector machine (SVM), developed by Vapnik and Chervonenkis [73], is one approach for building such a model.

Specifically, the SVM builds a *linear* model $f(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} + b$ when $\mathcal{X}$ is a vector space. We require that $f(\mathbf{x}_i) \geq 1$ for examples where $\mathbf{y}_i = 1$ and $f(\mathbf{x}_i) \leq -1$ for $\mathbf{y}_i = -1$. We also wish for $\|\mathbf{w}\|$ to be as small as possible, because this will generalize to new examples better than if we allow $\|\mathbf{w}\|$ to be large. Intuitively, if a new example $\mathbf{x}$ is perturbed by a small amount, then a model with a small $\|\mathbf{w}\|$ is less likely to move its prediction, which is the sign of $f(\mathbf{x})$, across the decision boundary to the other class.

With a convex objective, i.e., minimizing $\|\mathbf{w}\|$, and convex constraints, we can see that the SVM solves a convex program. Often the objective is given as $\frac{1}{2}\|\mathbf{w}\|_2^2$ so that it is differentiable, and sometimes as $\|\mathbf{w}\|_1$ to encourage sparsity (classifying on fewer features of the input). In the quadratic case, this is a quadratic program (QP):

$$\begin{aligned} \min \quad & \frac{1}{2}\|\mathbf{w}\|_2^2 \\ \text{s.t.} \quad & y_i(\mathbf{w}^\top \mathbf{x}_i + b) \geq 1,\ \forall i \in [1..n] \end{aligned} \tag{2.1}$$

The dual to this program is the following:

$$\begin{aligned} \max \quad & \sum_i \alpha_i - \frac{1}{2}\sum_i \sum_j \alpha_i \alpha_j y_i y_j \mathbf{x}_i^\top \mathbf{x}_j \\ \text{s.t.} \quad & \sum_i \alpha_i y_i = 0,\ \alpha_i \geq 0,\ \forall i \in [1..n] \end{aligned} \tag{2.2}$$

An SVM is also called a *maximum margin classifier* because it maximizes the space between positive and negative training examples. This form also has the drawback that if the training examples are not linearly classifiable, then an SVM cannot find a model that fits the training data. This form is called a *hard margin* SVM because there must not be any examples in the margin.

The margin can be "softened" by adding a loss function:

$$\min \quad \frac{1}{2}\|\mathbf{w}\|_2^2 + C\sum_{i=1}^{n}\ell(f(\mathbf{x}_i), y_i)$$

When the loss is the *hinge loss*, that is, $\ell(z, y) = \max(0, 1 - yz)$, this yields the formulation from Cortes and Vapnik [24]:

$$\begin{aligned} \min \quad & \frac{1}{2}\|\mathbf{w}\|_2^2 + C\sum_{i=1}^{n}\xi_i \\ \text{s.t.} \quad & y_i(\mathbf{w}^\top \mathbf{x}_i + b) \geq 1 - \xi_i, \, \forall i \in [1..n] \\ & \xi_i \geq 0 \end{aligned} \tag{2.3}$$

The squared hinge loss (replace ξ_i with ξ_i^2 in (2.3)) is also common. Soft-margin classifiers allow examples to lie inside the margin or even in the "wrong" part of the model. In many cases, this still trains a model that generalizes well. These forms of SVM also have dual forms, usually simple additions to the constraints on $\boldsymbol{\alpha}$.

Unfortunately, soft margins are still not enough to fit good models to some datasets. SVMs allow for a nice "trick" when the dataset does not allow for a good fit. We will address this issue in Section 2.3.

2.2.1 Modeling the Geometry of SVM

Alternatively, we can frame the SVM problem in a geometric way. Suppose that we have the same collection of n training examples in $\mathbb{R}^d \times \{-1, +1\}^n$ as above. In matrix form, $\mathbf{X} \in \mathbb{R}^{n \times d}$ (the rows $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$ are the examples) and $\mathbf{y} = (y_1, \ldots, y_n) \in \{-1, +1\}^n$ are the corresponding binary class labels for the data points in $\mathbf{X}$. Let $\mathbf{X}_+ \in \mathbb{R}^{n_+ \times d}$ denote the rows corresponding to the positive entries of $\mathbf{y}$, and likewise $\mathbf{X}_- \in \mathbb{R}^{n_- \times d}$ for the negative entries[1].

We can demonstrate that the dual SVM problem (2.2) 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 2.1**). 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

[1]The integers n_+ and n_-, where $n_+ + n_- = n$ merely indicate the counts of examples in their respective categories.

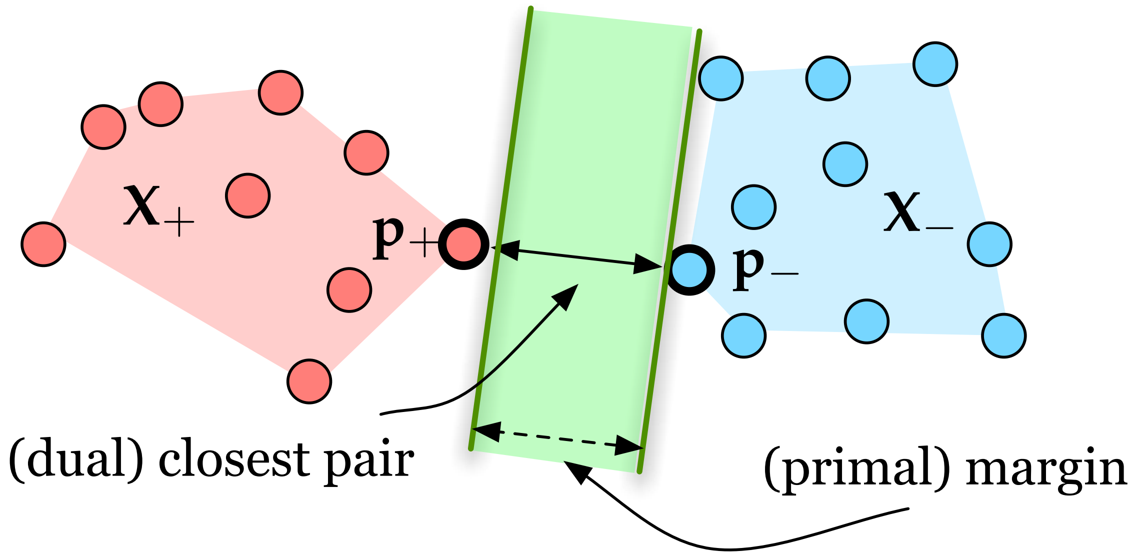

Figure 2.1: Illustration of primal-dual relationship for classification.

as $\boldsymbol{\alpha}_+^\top \mathbf{X}_+$, where $\boldsymbol{\alpha}_+^\top \mathbf{1} = 1$ and $\alpha_j \geq 0$, with a similar construction for $\mathbf{p}_-$ and $\boldsymbol{\alpha}_-$. This in turn can be written as an optimization:

$$\begin{aligned} \min_{\boldsymbol{\alpha}} \quad & \frac{1}{2}\|\mathbf{p}_+ - \mathbf{p}_-\|^2 \\ \text{s.t.} \quad & \boldsymbol{\alpha}_+^\top \mathbf{1} = 1,\ \boldsymbol{\alpha}_-^\top \mathbf{1} = 1,\ \boldsymbol{\alpha}_+, \boldsymbol{\alpha}_- \geq 0 \end{aligned} \tag{2.4}$$

Collecting all the α terms together by defining $\alpha_j \triangleq \alpha_{y_j,j}$, and expanding the distance term $\|\mathbf{p}_+ - \mathbf{p}_-\|^2$, it is straightforward to show that Problem (2.4) is equivalent to

$$\begin{aligned} \min_{\boldsymbol{\alpha}} \quad & \frac{1}{2}\boldsymbol{\alpha}^\top \mathbf{Y}\mathbf{X}\mathbf{X}^\top \mathbf{Y}\boldsymbol{\alpha} - \boldsymbol{\alpha}^\top \mathbf{1} \\ \text{s.t.} \quad & \boldsymbol{\alpha}^\top \mathbf{y} = 0,\ \alpha_i \geq 0, \end{aligned} \tag{2.5}$$

where (2.5) is the matrix-equivalent way of writing the familiar dual SVM problem (2.2). The equivalence of (2.4) and (2.2) is well known, so we decline to prove it here; see Bennett and Bredensteiner [10] for a proof of this equivalence.

2.3 Kernels and Reproducing Kernel Hilbert Spaces

If the soft margin technique is still not enough to give us a good model, then we need to either find a different technique altogether or extend SVMs to allow for richer models. One of the ways that we can accomplish the latter is to alter the space $\mathcal{X}$ in a way that allows for richer models, usually by increasing the dimensionality of the space.

We can construct a *lifting map* $\Phi : \mathcal{X} \to \mathcal{H}$ that adds these new dimensions. Φ is usually nonlinear, and one example is the map $\Phi : (x_1, x_2) \mapsto (x_1^2, \sqrt{2}x_1x_2, x_2^2)$. This map "lifts" the data from $\mathcal{X} = \mathbb{R}^2$ to $\mathcal{H} = \mathbb{R}^3$. In particular, this map allows us to classify a two-dimensional dataset where the labels are equal to $\operatorname{sgn}(x_1x_2)$. This labeling is not linearly classifiable in $\mathcal{X}$ and soft margins do not help much. In $\mathcal{H}$, however, the labeling is classifiable by the second coordinate alone. In the case where the labels are $\operatorname{sgn}(x_1^2 + x_2^2 - 1)$, the labeling is classifiable by the first and third coordinates in $\mathcal{H}$, and unclassifiable in $\mathcal{X}$.

If we make the restriction that $\mathcal{H}$ is a Hilbert space, then everything in programs (2.1), (2.4), and (2.2) still make sense. Obviously, this works when $\mathcal{H} = \mathbb{R}^d$, but this also works when $\mathcal{H}$ is infinite-dimensional. In this case, the dot product $(\cdot^\top \cdot)$ is replaced with the more general inner product $\langle \cdot, \cdot \rangle_\mathcal{H}$. The infinite-dimensional case is a problem though, because we cannot actually store $\Phi(\mathbf{x}_i)$, so we cannot get the SVM to produce a solution.

Looking at the dual problem (2.2) though, we can see that we never use the vector $\Phi(\mathbf{x}_i)$ except to take its dot product with another lifted vector, i.e., $\Phi(\mathbf{x}_i)^\top \Phi(\mathbf{x}_j)$. In an infinite-dimensional case, we would specify the inner product instead, i.e., $\langle \Phi(\mathbf{x}_i), \Phi(\mathbf{x}_j) \rangle_\mathcal{H}$. If we define the function $\kappa : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ as $\kappa : (\mathbf{x}_i, \mathbf{x}_j) \mapsto \langle \Phi(\mathbf{x}_i), \Phi(\mathbf{x}_j) \rangle_\mathcal{H}$, then we have something that we can potentially use. The dual program would become

$$\begin{aligned} \max \quad & \sum_i \alpha_i - \frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y_i y_j \kappa(\mathbf{x}_i, \mathbf{x}_j) \\ \text{s.t.} \quad & \sum_i \alpha_i y_i = 0,\ \alpha_i \geq 0,\ \forall i \in [1..n], \end{aligned} \tag{2.6}$$

or in matrix notation:

$$\begin{aligned} \max_{\boldsymbol{\alpha}} \quad & \boldsymbol{\alpha}^\top \mathbf{1} - \frac{1}{2} \boldsymbol{\alpha}^\top \mathbf{YKY} \boldsymbol{\alpha} \\ \text{s.t.} \quad & \boldsymbol{\alpha}^\top \mathbf{y} = 0,\ \alpha_i \geq 0,\ \forall i \in [1..n], \end{aligned} \tag{2.7}$$

where $\mathbf{K}$ is a matrix such that $K_{ij} = \kappa(\mathbf{x}_i, \mathbf{x}_j)$. This only works if $\mathbf{K}$ is symmetric and positive semidefinite ($\mathbf{K} \succeq 0$). $\mathbf{K}$ is called the *Gram matrix* for κ and $\mathbf{X}$.

If $\mathbf{K} \succeq 0$ for κ and *every* $\mathbf{X} \subset \mathcal{X}$, then κ is said to be a *positive-definite (p.d.) kernel*. Using a p.d. kernel in this way is called the *kernel trick* [15]. If κ has the *reproducing property* for a particular Hilbert space of functions $\mathcal{H}$, then $\langle f, \kappa(\cdot, \mathbf{y}) \rangle_\mathcal{H} = f(\mathbf{y})$ — i.e., $\kappa(\cdot, \mathbf{y})$ *evaluates* f at $\mathbf{y}$. Every p.d. kernel κ induces a unique Hilbert space of functions $\mathcal{H}$ for which κ has the reproducing property. This is known as the Moore-Aronszajn theorem [4] and $\mathcal{H}$ is called a *reproducing kernel Hilbert space (RKHS)*. The reproducing property turns out to be important for learning results.

2.3.1 Using Kernels to Predict

Kernels are useful for training a model, but there is one flaw: we do not know what that model should return if we pass in an example that we have not seen yet. We have so far based our model on values of κ evaluated at the training points and we have no idea about $\kappa(\mathbf{x}, \mathbf{x}_i)$. Fortunately, Schölkopf et al. [68] proved the *representer theorem*, which given a risk function like (2.6) shows that if a function $f^* \in \mathcal{H}$ satisfies (2.6), then f^* has a representation $f^*(\mathbf{x}) = \sum_{i=1}^{n} \alpha_i \kappa(\mathbf{x}, \mathbf{x}_i)$.

2.3.2 Kernelizing the SVM Dual

The geometric problem (2.4) also admits a kernelized form. 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* κ stands in for the inner product. This is dubbed the *kernel distance* [65] or the maximum mean discrepancy [36]. The dual formulation (2.5) 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 $\mathbf{G} \triangleq \mathbf{YKY}$:

$$\begin{aligned} \min_{\boldsymbol{\alpha}} \quad & \frac{1}{2}\boldsymbol{\alpha}^\top \mathbf{G} \boldsymbol{\alpha} - \boldsymbol{\alpha}^\top \mathbf{1} \\ \text{s.t.} \quad & \boldsymbol{\alpha}^\top \mathbf{y} = 0,\ \alpha_i \geq 0,\ \forall i \in [1..n]. \end{aligned} \tag{2.8}$$

Obviously, (2.8) is identical to program (2.7).

2.4 Kernel Learning

We can use any p.d. kernel we like, but we are left with the problem of selecting a good choice to begin with. *Kernel learning* is the problem of determining the best kernel (either from a dictionary of fixed kernels, or from a smooth space of kernel representations) for a given task. We could test several kernels with cross-validation and compare them, but this would take a lot of time. We would like a way to select the right kernel automatically.

Broadly speaking, we can divide kernel learning methods into two categories. The first, *multiple kernel learning (MKL)*, covers methods that largely assume that the desired kernel can be represented as a combination of a dictionary of *fixed* kernels, and seeks to learn their mixing weights. The other approach is based on a Fourier-analytic representation of shift-invariant kernels via Bochner's theorem [14]: roughly speaking, a kernel can be represented (in a Fourier dual form) as a probability distribution, and so the search for a kernel becomes a search over distributions.

In both approaches, training the model is challenging with many thousands of training points and hundreds of dimensions. Standard training approaches either employ some form of convex or alternating optimization (for MKL) or parameterize the space of distributions in terms of known distributions and try to optimize their parameters.

2.4.1 Multiple Kernel Learning

MKL 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. An early approach (uniformly weighted combination of kernels (UNIFORM), Pavlidis et al. [63]) eliminated the search and simply used an equal-weight sum of kernel functions. There are other MKL methods, but we will focus on those that extend SVMs.

In their seminal work, Lanckriet et al. [49] 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 SDP which in turn reduces to a QCQP. Soon after, Bach et al. [7] proposed a block-norm regularization method based on a second order cone program (SOCP).

For efficiency, researchers started using alternating optimization methods that alternate between updating the classifier parameters and the kernel weights. Sonnenburg et al. [69] modeled the MKL objective as a cutting plane problem and solved for kernel weights using Semi-Infinite Linear Programming (SILP) techniques. Rakotomamonjy et al. [67] used subgradient 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. [79]. Xu et al. [80] 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 [23], 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 [56], hyperkernels [61] and nonlinear families [25, 74]. Regularization based on the ℓ_2-norm [46] and ℓ_p-norm [45, 75] have also been introduced. In addition, stochastic gradient descent-based online algorithms for MKL have been studied in [62]. Another work by Jain et al. [40] discusses a scalable MKL algorithm for dynamic kernels. We briefly discuss and compare with this work when presenting empirical results (Section 3.4).

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. [26], who optimize the kernel weights in the first stage and learn a standard SVM in the second stage, and Kumar et al. [48], who train on meta-examples derived from kernel combinations on the ground examples. In Cortes et al. [26], 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 [18]. As for Kumar et al. [48], 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.

Following Lanckriet et al. [49], 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 $\mathrm{tr}(\mathbf{K}) = 1$. The dual problem then takes the following form [49]:

$$\begin{aligned} \max_{\mathbf{K}} \min_{\boldsymbol{\alpha}} \quad & \frac{1}{2}\boldsymbol{\alpha}^\top \mathbf{G}\boldsymbol{\alpha} - \boldsymbol{\alpha}^\top \mathbf{1} \\ \text{s.t.} \quad & \mathbf{K} = \sum_{i=1}^{m} \mu_i \mathbf{K}_i, \quad \mathrm{tr}(\mathbf{K}) = 1, \quad \mathbf{K} \succeq 0, \quad \boldsymbol{\mu} \geq 0 \end{aligned} \tag{2.9}$$

When juxtaposed with (2.4) and (2.2), this can be interpreted as searching for the kernel that maximizes the shortest (kernel) distance between polytopes.

2.4.2 Localized Multiple Kernel Learning

The rationale for *localized* kernel learning (as illustrated in Section 4.1) is to allow the weight assigned to different kernels to vary in different parts of the data space to incorporate any local structure in the data.

2.4.2.1 Localized Multiple Kernel Learning (LMKL)

Gönen and Alpaydin [33] were the first to propose an algorithm to solve this problem. They called their method *localized multiple kernel learning* (LMKL). The idea was to generalize the η_i to be functions of the data $\mathbf{x}$ as well as a set of *gating parameters* $\mathbf{V} \in \mathbb{R}^{d \times m}$.

They defined a gating function as:

$$\eta(\mathbf{x}|\mathbf{V}) = \text{softmax}(\mathbf{x}^\top \mathbf{V} + \mathbf{v}_0),$$

where $\mathbf{v}_0$ is an m-dimensional vector of offsets[2].

Given such a gating function, they then defined a generalized discriminant function:

$$f(\mathbf{x}) = \sum_{i=1}^{m} \eta_i(\mathbf{x}|\mathbf{V}) \langle w_i, \phi_i(\mathbf{x}) \rangle + b,$$

Expressing the classifier function leads to a non-convex optimization involving the parameters V. They then proposed solving this problem using a two-step alternating optimization algorithm, summarized in **Algorithm 1**.

Algorithm 1 LMKL

```
1: repeat
2:     Calculate $\mathbf{K}_\eta$, the Gram matrix of the combined kernel, with the gating functions $\eta_i$:
3:     $(K_\eta)_{jk} \leftarrow \kappa_\eta(\mathbf{x}_j, \mathbf{x}_k) = \sum_{i=1}^{m} \eta_i(\mathbf{x}_j)\kappa_i(\mathbf{x}_j, \mathbf{x}_k)\eta_i(\mathbf{x}_k)$
4:     Solve canonical SVM with $\mathbf{K}_\eta$
5:     Update gating parameters $\mathbf{V}$ using gradient descent
6: until convergence
```

The complexity of the overall algorithm is dominated by the time to perform the canonical SVM. Other variants of this basic framework include Yang et al. [81], which allows gating functions to operate on groups of points, and Han and Liu [37] which incorporates

[2]In later works, they proposed other gating functions that employed sigmoids and Gaussian functions [34].

a gating function based on pair-wise similarities inferred from a kernel density estimate for each kernel.

2.4.2.2 Convex LMKL (C-LMKL)

More recently, Lei et al. [52] noted the non-convex nature of the above objective function. In order to avoid the tendency of such functions to overfit to the training data, they proposed an alternate *convex* formulation of the localized multiple kernel learning problem. The central idea of their approach is to first construct a soft clustering of the data, represented by a soft assignment function $c_\ell(x_j)$ that associates point x_j with cluster ℓ. Next, they define parameters $\beta_{\ell i}$ that associate each of m kernels with each cluster ℓ: in effect, the soft clustering fixes the locality they wish to exploit, and the $\beta_{\ell i}$ then allow them to use different kernel combinations.

The resulting optimization is convex, assuming that the loss function is convex. This allows them to obtain generalization bounds as well as good prediction accuracy in practice. The optimization itself proceeds as a two-stage optimization: the first stage invokes a standard SVM solver to find the best weight vectors given the $\beta_{\ell i}$ and the second stage optimizes $\beta_{\ell i}$ for given weights. This latter stage can in fact be solved in closed form. Thus, as with LMKL, the term dominating the computation time is the use of an SVM solver.

2.4.2.3 Success-Based Locally-Weighted Kernels (SwMKL)

Kannao and Guha [44] introduced SwMKL as a way to localize kernel learning in a different manner. Their method is to analyze each kernel for its success on the input data, then construct a gating function based on smoothing the success with a regression, summarized in **Algorithm 2**.

Algorithm 2 SwMKL

```
1: for all $i \in [1..m]$ do
2:     Train classifier $f_i : \mathbb{R}^d \to \{-1, 1\}$ with kernel $\kappa_i$
3:     Train regressor $g_i : \mathbb{R}^d \to (0, 1)$ with $(\mathbf{X}, \delta(\mathbf{y}, f_i(\mathbf{X})))$
4: Train classifier using
```

$$\kappa(\mathbf{x}_j, \mathbf{x}_k) = \frac{\sum_{i=1}^{m} g_i(\mathbf{x}_j)\kappa_i(\mathbf{x}_j, \mathbf{x}_k)g_i(\mathbf{x}_k)}{\sum_{i=1}^{m} g_i(\mathbf{x}_j)g_i(\mathbf{x}_k)}$$

Its complexity is controlled by the initial SVM computations, the different support vector regression operations, as well as the final SVM calculation on the combined kernel function. The experimental approach in [44] is to separate each kernel by feature – essentially creating individual kernels for each combination of kernel and feature and then combining them. When testing with this algorithm, we had much better success when using a kernel on all features.

2.4.2.4 Sample-Adaptive Multiple Kernel Learning (SAMKL)

An alternate approach employed by Liu et al. [53] is to separate out the assignment of kernels to points and the weights associated with the kernels. In their formulation, which they describe as *sample-adaptive multiple kernel learning*, they introduce latent *binary* variables to decide whether a particular kernel should operate on a particular point or not. Each point is therefore mapped to a single point in the product of the feature spaces defined by the given kernels. Now they run a two-stage alternating optimization: in the first stage, given fixed values of the latent variables, they solve a multiple kernel learning problem for the different subspaces simultaneously, and then they run an integer program solver to obtain new values of the latent variables. Note that each step of the iteration here involves costly operations (an MKL solver and an integer program solver) in comparison with the SVM solvers in the other approaches.

2.5 Continuous Kernel Learning

Kernel learning is not limited to kernel mixtures, as described in the previous sections. Instead, we can speak about the set of *all* kernels (and RKHSs), and the process of selecting a kernel becomes a richer idea. Indeed, if we consider kernels as *analytical* objects, different tools present themselves for use.

2.5.1 Approaches Utilizing Bochner's Theorem

One of the key mathematical tools that drives much of kernel learning work is Bochner's theorem:

Bochner's theorem [14]. *A continuous function $k : \mathbb{R}^d \to \mathbb{R}$ is positive-definite*[3] *iff $k(\cdot)$ is the*

[3]For our purposes, we define k to be positive-definite if for any vectors $\{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$, the $n \times n$ matrix A,

Fourier transform of a non-negative measure.

Several papers have been published that explore the connection between Bochner's theorem [14] and learning a kernel[4]. A Bayesian view produces an interpretation of this optimization as learning the kernel of a Gaussian process (GP). Wilson and Adams [77] equate stationary (shift-invariant) kernels to the spectral density function of a GP. They observe that linear combinations of squared-exponential kernels are dense in the space of stationary kernels. The resulting kernel has few parameters and is relatively easy to interpret.

Yang et al. [84] extend the ideas in [77] and combine them with the principles from Fastfood [50]. The authors also discuss variants of their algorithms such as computing a piecewise linear kernel. Similarly, the BaNK method by Oliva et al. [60] learns a kernel using the GP technique and trains the kernel using MCMC. Finally in the GP vein, Wilson et al. [78] integrate a deep network as input to the GP, treating the GP as an "infinite-dimensional" layer of the network, and optimize the parameters of the GP simultaneously with the parameters of the network using backpropagation.

Băzăvan et al. [17], in contrast, optimize Fourier embeddings, but decompose each ω_i into a parameter σ_i multiplied by a nonlinear function of a uniform random variable to represent the sample. The uniform variable is resampled during optimization as the parameter is learned.

2.5.2 Infinite-width Networks

Early work on infinite-width networks was done by Neal [59], who tied infinite networks to Gaussian processes, assuming that the distribution is Gaussian. Cho and Saul [21] analyzed the case where the network is either a step network (the output is 1 if the input is positive, 0 otherwise) or a rectified linear unit (ReLU), a type of network used frequently in deep networks (the input z is passed through the function $\max\{0, z\}$). They showed that if the distribution is Gaussian in these settings, the function $\phi_{\mathbf{x}}$ output by the network is a lifting map corresponding to a kernel they dub the *arc-cosine kernel*. Hazan and Jaakkola

where $a_{ij} = k(\mathbf{x}_i - \mathbf{x}_j)$, is positive semidefinite. That is, $y^\top Ay \geq 0$ for any $y \in \mathbb{R}^n$.

[4]Note that Yang et al. [83] are not producing a kernel learning method, but an effective way to sparsify CNNs. No comparison to other kernel learning methods is made in [83].

[39] extended this result further, and analyzed the kernel corresponding to two infinite layers stacked in series. They showed that such a network, when the distribution of the first layer is Gaussian, and the second layer is treated as a Gaussian process (a process is a distribution of distributions), corresponds to a kernel that can be computed explicitly. Globerson and Livni [32] produce an online algorithm for infinite-layer networks that *avoids* the kernel trick. They demonstrate a sample complexity equal to methods that use the kernel trick, demonstrating that sampling can be as effective as methods that have access to kernel values.

2.5.3 Layered Kernels

Zhuang et al. [87] develop a multiple kernel learning technique where they use a layered kernel to combine the output of several other kernels. Their algorithm alternates the use of standard SVM and stochastic gradient descent. Lu et al. [54] scale up [66] by making some interesting mathematical observations about kernels and distributions. Their work relies heavily on the correspondence between distributions and kernels, a theme that we explore as well. Yu et al. [86] also seek to optimize a kernel, using alternating optimization and also based on Bochner's theorem [14]. Jiu and Sahbi [41, 42] exploit *kernel map networks* and Laplacians of nearest-neighbor graphs [42] to produce "deep" kernels for use in SVMs.

2.5.4 Neural Networks as Kernels

Yang et al. [83] exploit the correspondence between ReLUs and arc-cosine kernels [21], and the sparsity of the Fastfood transform [50] to reduce the complexity of a convolutional neural net.

Aslan et al. [6] seek to make the optimization of neural networks convex through kernels and matrix techniques. Mairal et al. [55] extend hierarchical kernel descriptors [12, 13] to act as convolutional layers. Very recently, Wilson et al. [78] combine neural networks with Gaussian processes, drawing on the infinite-width network setting, to produce "deep" kernels.

PART I

KERNEL LEARNING IN SVMS

CHAPTER 3

MULTIPLICATIVE WEIGHT UPDATES-MULTIPLE KERNEL LEARNING

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 datasets 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.

3.1 Introduction

Multiple kernel learning is a principled alternative to choosing kernels, and has been successfully applied to a wide variety of learning tasks and domains [3, 7, 27, 49, 63, 69, 85, 88]. Pioneering work by Lanckriet et al. [49] 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 [49, 67, 69, 79, 80], albeit accurate, are slow and have large memory footprints.

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 (see (2.9)). 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 and Kale [5], a primal-dual combinatorial algorithm for solving semidefinite programs (SDPs). 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 [69] that use careful engineering to parallelize MKL optimizations in order to scale can be viewed as complementary to our work.

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 [28] after Nyström kernel approximations are applied as feature transformations, and (c) compares favorably with the uniformly weighted combination of kernels (UNIFORM) heuristic that merely averages all kernels without searching for an optimal combination. As has been noted [23], the UNIFORM heuristic is a strong baseline for the evaluation of MKL methods. We use LibLinear with Nyström kernel approximations (LIBLINEAR+) as an additional scalable baseline, and we are able to beat both these baselines when both m and n are significantly large.

3.2 Our Algorithm

The MKL formulation of (2.9) can be transformed (as we shall see later) into a QCQP that can be solved by a number of different solvers [2, 49, 70]. 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 (2.9) 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 and n, and (b) it is not obvious how to adjust kernel weights in each iteration.

3.2.1 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 and Kale [5]. 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 $O(mn)$.

3.2.2 QCQPs and SDPs

We start by using an observation due to Lanckriet et al. [49] to convert (2.9)[1] into the following QCQP:

$$\begin{aligned} \max_{\boldsymbol{\alpha},s} \quad & (2\boldsymbol{\alpha}^\top \mathbf{1} - s) \\ \text{s.t.} \quad & s \geq \frac{1}{r_i}\boldsymbol{\alpha}^\top \mathbf{G}_i \boldsymbol{\alpha}, \quad \boldsymbol{\alpha}^\top \mathbf{y} = 0, \quad \boldsymbol{\alpha} \geq 0 \end{aligned} \tag{3.1}$$

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

Next, we rewrite (3.1) in canonical SDP form in order to apply the MMWU framework:

$$\begin{aligned} \omega^* = \max_{\boldsymbol{\alpha},s} \quad & 2\boldsymbol{\alpha}^\top \mathbf{1} - s \\ \text{s.t.} \quad & \forall i \in [1..m] \quad \mathbf{Q}_i(\boldsymbol{\alpha}) = \begin{pmatrix} \mathbf{I}_n & \mathbf{A}_i\boldsymbol{\alpha} \\ (\mathbf{A}_i\boldsymbol{\alpha})^\top & s \end{pmatrix}, \\ & \mathbf{Q}_i(\boldsymbol{\alpha}) \succeq \mathbf{0}, \quad \boldsymbol{\alpha}^\top \mathbf{y} = 0, \quad \boldsymbol{\alpha} \geq \mathbf{0}. \end{aligned} \tag{3.2}$$

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

[1]We note that (3.1) is the *hard-margin* version of the MKL problem. The standard soft-margin variants can also be placed in this general framework [49]. For the 1-norm soft margin, we add the constraint that all terms of $\boldsymbol{\alpha}$ are upper bounded by the margin constant C. For the 2-norm soft margin, another term $\frac{1}{C}\boldsymbol{\alpha}^\top\boldsymbol{\alpha}$ appears in the objective, or we can simply add a constant multiple of $\mathbf{I}$ to each $\mathbf{G}_i$.

3.2.3 The MMWU Framework

We give a brief overview of the MMWU framework of Arora and Kale [5] (for more details, the reader is directed to Satyen Kale's thesis [43]). The approach starts with a "guess" ω for the optimal value ω^* 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 3 MMWU template [5]

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

The process starts with some assignment to $\mathbf{P}^{(1)}$ (typically the identity matrix **I**). If this assignment is both primal feasible and at most ω, the process ends. Else, there must be some assignment to $\boldsymbol{\alpha}$ (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 ω, and satisfies constraints (3.1)).

The primal constraints and $\boldsymbol{\alpha}$ are then used to guide the search for a new primal assignment $\mathbf{P}^{(t+1)}$. They are combined to form the matrix $\mathbf{Q}_i(\boldsymbol{\alpha}^{(t)})$ (see (3.1)), and then adjusted to form an "event matrix" $\mathbf{M}^{(t)}$ (see Section 3.2.4.1 for details)[2]. Exponentiating the sum of all the observed $\mathbf{M}^{(t)}$ so far, the algorithm exponentially reweights primal constraints that are more important, and the process repeats. By minimizing the loss, the assignments to $\mathbf{P}^{(t)}$ and $\boldsymbol{\alpha}^{(t)}$ are guaranteed to result in an SDP value that approximates ω^* within a factor of $(1+\epsilon)$.

[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.

3.2.4 Our Algorithm

We now adapt the above framework to solve the MKL SDP given by (3.2). As we will explain below, we can assign ω^* *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 $\boldsymbol{\alpha}^{(t)}$, 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 μ_i. We denote $\boldsymbol{\alpha} = \sum_i \boldsymbol{\alpha}^{(i)}$ as the accumulated dual assignment thus far and $\mathbf{M}_i = \sum_t \mathbf{M}_i^{(t)}$ as the accumulated i^{th} event matrix.

3.2.4.1 The Backward Step

It will be convenient to explain the backward step first. Given $\boldsymbol{\alpha}^{(t)}$ and $\mathbf{Q}_i(\boldsymbol{\alpha}^{(t)})$, we define $\mathbf{M}_i^{(t)} \triangleq \frac{1}{2\rho}(\mathbf{Q}_i(\boldsymbol{\alpha}^{(t)}) + \rho \mathbf{I}_{n+1})$ where ρ 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 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*

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

Proof. We symbolically exponentiate an $n+1 \times n+1$ matrix of the form

$$\mathbf{M} = \begin{pmatrix} a\mathbf{I}_n & \mathbf{u} \\ \mathbf{u}^\top & a \end{pmatrix}.$$

Since this matrix is real and symmetric, its eigenvalues λ_i are positive and its unit eigenvectors $\mathbf{v}_i$ form an orthonormal basis. The method that we use to symbolically exponentiate it is to express it in the form

$$\mathbf{M} = \sum_{i=1}^{n} \lambda_i \mathbf{v}_i \mathbf{v}_i^\top.$$

The exponential then becomes

$$e^{\mathbf{M}} = \sum_{i=1}^{n} e^{\lambda_i} \mathbf{v}_i \mathbf{v}_i^\top.$$

3.2.4.1.1 Eigenvalues. The characteristic polynomial for $\mathbf{M}$ is not difficult to calculate. It is:

$$(\lambda - a)^{n-1}(\lambda^2 - 2a\lambda + a^2 - \|\mathbf{u}\|^2) = (\lambda - a)^{n-1}(\lambda - a + \|\mathbf{u}\|)(\lambda - a - \|\mathbf{u}\|).$$

This yields $n-1$ eigenvalues equal to a, and the other two equal to $a+\|\mathbf{u}\|$ and $a-\|\mathbf{u}\|$. We label them λ_1 and λ_2, respectively, and the rest are equal to a.

3.2.4.1.2 Eigenvectors. First we show that $\mathbf{M}$ has two specific eigenvectors:

$$\begin{pmatrix} a\mathbf{I}_n & \mathbf{u} \\ \mathbf{u}^\top & a \end{pmatrix} \begin{pmatrix} \mathbf{u} \\ \pm\|\mathbf{u}\| \end{pmatrix} = \begin{pmatrix} (a\pm\|\mathbf{u}\|)\mathbf{u} \\ \|\mathbf{u}\|^2 \pm a\|\mathbf{u}\| \end{pmatrix} = (a\pm\|\mathbf{u}\|) \begin{pmatrix} \mathbf{u} \\ \pm\|\mathbf{u}\| \end{pmatrix},$$

so these are eigenvectors of the form $(\mathbf{u}, \pm\|\mathbf{u}\|)^\top$ with eigenvalues $a\pm\|\mathbf{u}\|$. We will call the corresponding eigenvectors $\mathbf{v}_1$ and $\mathbf{v}_2$. Since $\mathbf{M}$ is symmetric, all of its eigenvectors are orthogonal. The remaining eigenvectors are of the form $(\mathbf{w}, 0)^\top$, where $\mathbf{w}^\top\mathbf{u} = 0$:

$$\begin{pmatrix} a\mathbf{I}_n & \mathbf{u} \\ \mathbf{u}^\top & a \end{pmatrix} \begin{pmatrix} \mathbf{w} \\ 0 \end{pmatrix} = \begin{pmatrix} a\mathbf{w} \\ 0 \end{pmatrix}.$$

Clearly the corresponding eigenvalue for any such eigenvector is a, so there are $n-1$ of them. The corresponding parts of these eigenvectors are labeled $\mathbf{w}_i$, where $3 \le i \le n+1$, and we assume they are unit vectors.

3.2.4.1.3 The exponential. For unit eigenvectors $\hat{\mathbf{v}}_i$, since

$$e^{\mathbf{M}} = \sum_{i=1}^{n} e^{\lambda_i} \frac{\mathbf{v}_i\mathbf{v}_i^\top}{\|\mathbf{v}_i\|^2},$$

and the eigenvalue a is of multiplicity $n-1$, we have

$$\begin{aligned} e^{\mathbf{M}} &= \frac{e^{\lambda_1}}{2\|\mathbf{u}\|^2} \begin{pmatrix} \mathbf{u}\mathbf{u}^\top & \|\mathbf{u}\|\mathbf{u} \\ \|\mathbf{u}\|\mathbf{u}^\top & \|\mathbf{u}\|^2 \end{pmatrix} + \frac{e^{\lambda_2}}{2\|\mathbf{u}\|^2} \begin{pmatrix} \mathbf{u}\mathbf{u}^\top & -\|\mathbf{u}\|\mathbf{u} \\ -\|\mathbf{u}\|\mathbf{u}^\top & \|\mathbf{u}\|^2 \end{pmatrix} + e^a \sum_{i=3}^{n} \begin{pmatrix} \mathbf{w}_i\mathbf{w}_i^\top & \mathbf{0} \\ \mathbf{0}^\top & 0 \end{pmatrix} \\ &= e^a \left[\frac{e^{\|\mathbf{u}\|}}{2} \begin{pmatrix} \hat{\mathbf{u}}\hat{\mathbf{u}}^\top & \hat{\mathbf{u}} \\ \hat{\mathbf{u}}^\top & 1 \end{pmatrix} + \frac{e^{-\|\mathbf{u}\|}}{2} \begin{pmatrix} \hat{\mathbf{u}}\hat{\mathbf{u}}^\top & -\hat{\mathbf{u}} \\ -\hat{\mathbf{u}}^\top & 1 \end{pmatrix} + \begin{pmatrix} \mathbf{I}_n - \hat{\mathbf{u}}\hat{\mathbf{u}}^\top & \mathbf{0} \\ \mathbf{0}^\top & 0 \end{pmatrix} \right] \\ &= e^a \left[\begin{pmatrix} \cosh\|\mathbf{u}\|\hat{\mathbf{u}}\hat{\mathbf{u}}^\top & \sinh\|\mathbf{u}\|\hat{\mathbf{u}} \\ \sinh\|\mathbf{u}\|\hat{\mathbf{u}}^\top & \cosh\|\mathbf{u}\| \end{pmatrix} + \begin{pmatrix} \mathbf{I}_n - \hat{\mathbf{u}}\hat{\mathbf{u}}^\top & \mathbf{0} \\ \mathbf{0}^\top & 0 \end{pmatrix} \right]. \end{aligned}$$

The last term in the equality is due to the fact that $\hat{\mathbf{u}}$ and the $\hat{\mathbf{w}}_i$ form an orthonormal basis for $\mathbb{R}^n$, so $\hat{\mathbf{u}}\hat{\mathbf{u}}^\top + \sum \hat{\mathbf{w}}_i\hat{\mathbf{w}}_i^\top = \mathbf{I}_n$. □

Lemma 1 implies that we can exponentiate the event matrix $\mathbf{M}^{(t)}$ (see **Algorithm 3**) quickly, as promised. In particular, we set $\mathbf{P}_i^{(t+1)} = c\exp(-\varepsilon\sum_t \mathbf{M}_i^{(t+1)})$ where c normalizes the matrix to have unit trace.

In **Lemma 1**, large inputs to the functions exp, cosh, and sinh will cause them to rapidly overflow even at double-precision range. Fortunately, there are two steps we can take. First, $\cosh(x)$ and $\sinh(x)$ converge exponentially to $\exp(x)/2$, so above a high enough value, we can simply approximate $\sinh(x)$ and $\cosh(x)$ with $\exp(x)/2$.

Because exp can overflow just as much as sinh or cosh, this does not solve the problem completely. However, since $\mathbf{P}$ is always normalized so that $\text{tr}(\mathbf{P}) = 1$, we can multiply the elements of $\mathbf{P}$ by any factor we choose and the factor will be normalized out in the end. So above a certain value, we can use exp alone and throw a "quashing" factor ($e^{-\phi-q}$) into the equations before computing the result, and it will be normalized out later in the computation (this also means that we can ignore the e^{a} factor). For our purposes, setting $q = 20$ suffices. This trades overflow for underflow, but underflow can be interpreted merely as one kernel disappearing from significance.

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 $\mathbf{P}_i^{(t)}$.

3.2.4.1.4 The exponentiation algorithm. From $\mathbf{M}_i^{(t)}$ in **Algorithm 3** and (3.2), we have $\mathbf{M}_i^{(t)} = \frac{1}{2\rho}(\mathbf{Q}_i(\boldsymbol{\alpha}^{(t)}) + \rho\mathbf{I}_{n+1})$, where ρ is a program parameter which is explained in Section 3.2.7.

Our $\mathbf{Q}_i(\boldsymbol{\alpha}) = \begin{pmatrix} \mathbf{I}_n & \mathbf{A}_i\boldsymbol{\alpha} \\ (\mathbf{A}_i\boldsymbol{\alpha})^\top & 1 \end{pmatrix}$ is of the form $\begin{pmatrix} a\mathbf{I}_n & \mathbf{u}_i \\ \mathbf{u}_i^\top & a \end{pmatrix}$, where $a = 1\ \forall i$ and $\mathbf{u}_i = \mathbf{A}_i\boldsymbol{\alpha}$. So we have

$$\mathbf{u}_i^\top\mathbf{u}_i = (\mathbf{A}_i\boldsymbol{\alpha})^\top\mathbf{A}_i\boldsymbol{\alpha} = \boldsymbol{\alpha}^\top\mathbf{A}_i^\top\mathbf{A}_i\boldsymbol{\alpha} = \boldsymbol{\alpha}^\top\frac{1}{r_i}\mathbf{G}_i\boldsymbol{\alpha},$$

where the last equality follows from $\mathbf{A}_i^\top\mathbf{A}_i = \frac{1}{r_i}\mathbf{G}_i$ (cf. (3.2)). As we shall show in **Algorithm 6**, at each iteration the matrix to be exponentiated is a sum of matrices of the form $\frac{1}{2\rho}(\mathbf{Q}_i(\sum_{t=1}^{\tau}\boldsymbol{\alpha}^{(t)}) + \rho t\mathbf{I}_{n+1})$, so **Lemma 1** can be applied at every iteration.

We provide in detail the algorithm we use to exponentiate the matrix $\mathbf{M}$ in **Algorithm 4**. Note that the algorithm "warms up" until the quashing term q is large enough, and then (smoothly) swaps over to what is essentially softmax, or standard MWU. Note that we elide the quashing computation in the overflow case, because most softmax implementations will do this internally.

3.2.4.2 The Forward Step

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

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

Algorithm 4 Exponentiate-M

```
Input: $\mathbf{y}$, $\boldsymbol{\alpha}$, $\{\mathbf{G}_i\}$, $\varepsilon'$, $\rho$
  for $i \in [1..m]$ do
      $\|\mathbf{u}_i\| \leftarrow \sqrt{\boldsymbol{\alpha}^\top \mathbf{G}_i \boldsymbol{\alpha}}$
      $\mathbf{g}_i \leftarrow \frac{1}{\|\mathbf{u}_i\|} \mathbf{G}_i \boldsymbol{\alpha}$
      $\|\mathbf{u}_i\| \leftarrow \frac{\varepsilon'}{2\rho} \|\mathbf{u}_i\|$
  $q \leftarrow \max_i \|\mathbf{u}_i\|$
  if $q < 20$ then
      for $i \in [1..m]$ do
          $p_i^{11} \leftarrow 2\cosh(\|\mathbf{u}_i\|)$
          $p_i^{12} \leftarrow 2\sinh(\|\mathbf{u}_i\|)$
      $S \leftarrow m(n-1) + \sum_{i=1}^m p_i^{11}$
      for $i \in [1..m]$ do
          $p_i^{12} \leftarrow -p_i^{12}/S$
  else
      $\mathbf{p}^{12} \leftarrow -\operatorname{softmax}(\|\mathbf{u}_i\|)_{i=1}^m$
  $\mathbf{g} \leftarrow \sum_i p_i^{12} \mathbf{g}_i$
  return $\mathbf{p}^{12}$, $\mathbf{g}$
```

The existence of such a $\boldsymbol{\alpha}^{(t)}$ will prove that the current guess $\mathbf{P}^{(t)}$ is either primal infeasible or suboptimal (see Arora and Kale [5] for details).

We now exploit the structure of $\mathbf{P}^{(t)}$ given by **Lemma 1**. In particular, let $p_i^{11} = p_i^{22} = e^a \cosh \|\mathbf{u}_i\| / \operatorname{tr} \mathbf{P}$ and $p_i^{12} = -e^a \sinh \|\mathbf{u}_i\| / \operatorname{tr} \mathbf{P}$. So

$$\mathbf{Q}_i(\boldsymbol{\alpha}^{(t)}) \bullet \mathbf{P}_i = \begin{pmatrix} 0 & \mathbf{A}_i \boldsymbol{\alpha}^{(t)} \\ (\mathbf{A}_i \boldsymbol{\alpha}^{(t)})^\top & 0 \end{pmatrix} \bullet \mathbf{P}_i + \mathbf{I}_{n+1} \bullet \mathbf{P}_i = 2 p_i^{12} \hat{\mathbf{u}}_i^\top \mathbf{A}_i \boldsymbol{\alpha}^{(t)} + \operatorname{tr}(\mathbf{P}_i)$$

$\sum_i \mathbf{Q}_i(\boldsymbol{\alpha}^{(t)}) \bullet \mathbf{P}_i \geq 0$ then reduces to:

$$(\boldsymbol{\alpha}^{(t)})^\top \sum_{i=0}^{m} (2 p_i^{12} \mathbf{A}_i \hat{\mathbf{u}}_i) \geq -\operatorname{tr}(\mathbf{P}).$$

The right-hand side is the negative trace of $\mathbf{P}$ (which is normalized to 1), so this becomes

$$(\boldsymbol{\alpha}^{(t)})^\top \sum_i 2 p_i^{12} \mathbf{g}_i \geq -1, \tag{3.3}$$

where $\mathbf{g}_i = (\frac{1}{r_i} \mathbf{G}_i \boldsymbol{\alpha}) / (\frac{1}{r_i} \boldsymbol{\alpha}^\top \mathbf{G}_i \boldsymbol{\alpha})^{1/2}$. If we let $\mathbf{g} = \sum_i 2 p_i^{12} \mathbf{g}_i$ (which can be calculated at the end of the *backward* step), then we have simply $\mathbf{g}^\top \boldsymbol{\alpha} \geq -1$ which is a simple collection of linear constraints that can *always* be satisfied[3].

[3]The current margin borders a convex combination of points from each side. If we could not find a point

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 $\boldsymbol{\alpha}$ that satisfies (3.3), we simply choose the highest elements of $\mathbf{g}$ that correspond to both positive and negative labels, then set each corresponding entry in $\boldsymbol{\alpha}$ to $\frac{1}{2}$. **Algorithm 5** describes the pseudo-code for this process.

Algorithm 5 FIND-$\boldsymbol{\alpha}$

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

We highlight two important practical consequences of our formulation. First, the procedure produces a very sparse update to $\boldsymbol{\alpha}$: in each iteration, only two coordinates of $\boldsymbol{\alpha}$ are updated. This makes each iteration very efficient, taking only linear time. Second, by expressing $\mathbf{u}_i$ in terms of $\mathbf{g}_i$, we never need to explicitly compute $\mathbf{A}_i$ (as $\mathbf{u}_i = \mathbf{A}_i \boldsymbol{\alpha}$), which in turn means that we do not need to compute the (expensive) square root of $\mathbf{G_i}$ explicitly.

Another beneficial feature of the dual-finding procedure for MKL is that terms involving the primal variables $\mathbf{P}$ are either normalized (when we set the trace of $\mathbf{P}$ to 1) or eliminated (due to the fact that we have a compact closed-form expression for $\mathbf{P}$), *which means that we never have to explicitly maintain* $\mathbf{P}$, save for a small number ($4m$) of variables.

3.2.5 Avoiding Binary Search for ω

The objective function in (3.2) is *linear*, so we can scale s and $\boldsymbol{\alpha}$ and use the fact that $s = \boldsymbol{\alpha}^\top \mathbf{1} = \omega$ to transform the problem[4]:

such that the inequality is satisfied, then no point from the convex combination can be found on or past the margin, which is impossible.

[4]This fact follows from the KKT conditions for the original problem. The support constraints of the SVM problem can be written as $\mathbf{G}\boldsymbol{\alpha} + b\mathbf{y} \geq \mathbf{1}$. If we multiply both sides of this inequality by $\boldsymbol{\alpha}^\top$, then it becomes an equality (by complementary slackness): $\boldsymbol{\alpha}^\top \mathbf{G}\boldsymbol{\alpha} = \boldsymbol{\alpha}^\top \mathbf{1}$. s is a substitution for $\boldsymbol{\alpha}^\top \mathbf{G}\boldsymbol{\alpha}$ in the MKL problem [49] so $s = \boldsymbol{\alpha}^\top \mathbf{1} = \omega$ as well.

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

The first constraint can be transformed back into an optimization; that is,

$$\begin{aligned} \min_{\omega} \max_{\boldsymbol{\alpha},i} \quad & \frac{1}{r_i}\boldsymbol{\alpha}^\top \mathbf{G}_i\boldsymbol{\alpha} \\ \text{s.t.} \quad & \boldsymbol{\alpha}^\top \mathbf{y} = 0, \quad \boldsymbol{\alpha}^\top \mathbf{1} = 1, \quad \boldsymbol{\alpha} \geq 0. \end{aligned}$$

Because ω does not figure into the maximization, we can compute ω simply by maximizing $\frac{1}{r_i}\boldsymbol{\alpha}^\top \mathbf{G}_i\boldsymbol{\alpha}$. Practically, this means that we simply add the constraint $\boldsymbol{\alpha}^\top \mathbf{1} = 1$, and the "guess" for ω is set to 1. We then *know* the objective, and only one iteration is needed, so the binary search is eliminated.

3.2.6 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

$$\sum_{i=1}^{m} \frac{2p_i^{12}}{r_i} \left(\frac{r_i}{\boldsymbol{\alpha}^\top \mathbf{G}_i \boldsymbol{\alpha}} \right)^{1/2} \boldsymbol{\alpha}^\top \mathbf{G}_i \boldsymbol{\alpha} = 1. \tag{3.4}$$

Now recall (from section Section 3.2.4.1) that $\boldsymbol{\alpha}^\top \mathbf{G}\boldsymbol{\alpha} = \sum_{i=1}^{m} \mu_i \cdot \boldsymbol{\alpha}^\top \mathbf{G}_i\boldsymbol{\alpha}$, and we also use the fact that $\boldsymbol{\alpha}^\top \mathbf{G}\boldsymbol{\alpha} = \boldsymbol{\alpha}^\top \mathbf{1} = \omega = 1$. Combining the above two, we have:

$$\sum_{i=1}^{m} \mu_i \cdot \boldsymbol{\alpha}^\top \mathbf{G}_i \boldsymbol{\alpha} = 1 \tag{3.5}$$

Matching (3.4) with (3.5) suggests that $\frac{2p_i^{12}}{r_i} \left(\frac{r_i}{\boldsymbol{\alpha}^\top \mathbf{G}_i \boldsymbol{\alpha}} \right)^{1/2}$ is the appropriate choice for μ_i.

3.2.7 Putting It All Together

Algorithm 6 summarizes the discussion in this section. The parameter ε is the error in approximating the objective function, but its connection to classification accuracy is loose. We set the actual value of ε via cross-validation (see Section 3.4). The parameter ρ is the *width* of the SDP, a parameter that indicates how much the solution can vary at each step. ρ is equal to the maximum absolute value of the eigenvalues of $\mathbf{Q}_i(\boldsymbol{\alpha}^{(t)})$, for any i [5].

Lemma 2. *ρ is bounded by* $3/2$.

Proof. ρ is defined as the maximum of $\|\mathbf{Q}(\boldsymbol{\alpha}^{(t)})\|$ for all t. Here $\|\cdot\|$ denotes the largest eigenvalue in absolute value [5]. Because $s = \omega = 1$ (see Section 3.2), the eigenvalues of $\mathbf{Q}_i(\boldsymbol{\alpha}^{(t)})$ are 1 (with multiplicity $n-1$), and $1 \pm \|\mathbf{A}_i\boldsymbol{\alpha}^{(t)}\|$. The greater of these in absolute value is clearly $1 + \|\mathbf{A}_i\boldsymbol{\alpha}^{(t)}\|$.

$\|\mathbf{A}_i\boldsymbol{\alpha}^{(t)}\|$ is equal to

$$((\boldsymbol{\alpha}^{(t)})^T\mathbf{A}_i^T\mathbf{A}_i\boldsymbol{\alpha}^{(t)})^{\frac{1}{2}} = \left(\frac{1}{r_i}(\boldsymbol{\alpha}^{(t)})^T\mathbf{G}_i\boldsymbol{\alpha}^{(t)}\right)^{\frac{1}{2}}.$$

$\boldsymbol{\alpha}^{(t)}$ always has two nonzero elements, and they are equal to $\frac{1}{2}$. They also correspond to values of $\mathbf{y}$ with opposite signs, so if j and k are the coordinates in question, $(\boldsymbol{\alpha}^{(t)})^T\mathbf{G}_i\boldsymbol{\alpha}^{(t)} \leq (1/4)(\mathbf{G}_{i(jj)} + \mathbf{G}_{i(kk)})$, because $\mathbf{G}_{i(jk)}$ and $\mathbf{G}_{i(kj)}$ are both negative.

Because of the factor of $1/r_i$, and because $r_i = \operatorname{tr}\mathbf{G}_i$, $\|\mathbf{A}_i\boldsymbol{\alpha}^{(t)}\| \leq \frac{1}{2}$. This is true for any of the i, so the maximum eigenvalue of $\mathbf{Q}(\boldsymbol{\alpha}^{(t)})$ in absolute value is bounded by $1 + \frac{1}{2} = \frac{3}{2}$. □

Note from the proof that since $\|\mathbf{A}_i\boldsymbol{\alpha}^{(t)}\| \leq \frac{1}{2}$, this also means that the eigenvalues of $\mathbf{Q}(\boldsymbol{\alpha}^{(t)})$ are bounded *below* by $\frac{1}{2}$. This has consequences for the running time of our algorithm.

3.2.7.1 Running Time

Every iteration of **Algorithm 6** will require a call to FIND-$\boldsymbol{\alpha}$, a call to EXPONENTIATE-M, and an update to $\mathbf{G}_i\boldsymbol{\alpha}$ and $\boldsymbol{\alpha}^\top\mathbf{G}_i\boldsymbol{\alpha}$. FIND-$\boldsymbol{\alpha}$ 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-$\boldsymbol{\alpha}$.

Algorithm 6 requires a total of T iterations at most, where $T = \frac{4\rho}{\varepsilon}\ln(n)$. Because the eigenvalues are guaranteed to be positive, we can use the $\ell \leq 0$ case described in Section 3.5. Since we only require one run of the main algorithm, the running time is bounded by

$$O\left(mn\ln(n)\frac{1}{\varepsilon}\right).$$

3.3 Single-Kernel Form

Interestingly, our proposed MWUMKL can easily be run as a *single*-kernel algorithm. This simplifies the algorithm considerably, since reweighting kernels is unnecessary.

Algorithm 6 MWUMKL

```
Input: $\mathbf{g}^{(1)} = \mathbf{0}$;
  $\rho$, the width;
  $\varepsilon$, the desired approximation error
  Set $\varepsilon' = \ln(\frac{1}{2})$
  Set $T = \frac{4\rho}{\varepsilon}\ln(n)$
  repeat[$T$ times]
      Get $\boldsymbol{\alpha}^{(t)}$ from Algorithm 5
      if Algorithm 5 failed then
          return
      Update $\boldsymbol{\alpha} = \boldsymbol{\alpha} + \boldsymbol{\alpha}^{(t)}$
      Set $\mathbf{M}_i^{(t)} = \frac{1}{2\rho}\left(\mathbf{Q}_i(\boldsymbol{\alpha}^{(t)}) + \rho\mathbf{I}_{n+1}\right)$
      Set $\mathbf{W}_i^{(t)} = (\frac{1}{2})^{\sum_{t=1}^{T}\mathbf{M}_i^{(t)}} = e^{\varepsilon'\sum_{t=1}^{T}\mathbf{M}_i^{(t)}}$ (Algorithm 4)
      Set $\mathbf{P}_i^{(t+1)} = \mathbf{W}_i^{(t)} / \operatorname{tr}(\mathbf{W}_i^{(t)})$
      Compute $\mathbf{g}^{(t+1)}$ from $\mathbf{P}^{(t+1)}$, $\{\mathbf{G}_i\}$, and $\boldsymbol{\alpha}$
  until $t = T$
  return $\frac{1}{T}\boldsymbol{\alpha}$, $\mathbf{P}^{(T+1)}$
```

3.3.1 Polytope Distance Problem

Gärtner and Jaggi [30] describe the polytope distance problem as that of finding the closest point on the convex hull of a set of points to the origin. When the points are the set of all difference vectors $\mathbf{u}_{ij} = \mathbf{x}_i^+ - \mathbf{x}_j^-$ for all $(i, j) \in [n^+] \otimes [n^-]$ between positive and negative examples, the problem is equivalent to solving an SVM. Gilbert's algorithm [31] proceeds by iteratively picking points that are closer to the origin than the current point, along the axis between the origin and the current point, and averaging them into the next current point.

Note that when proceeding this way with the SVM-equivalent points, all that needs to happen is to choose the most violating point from each class. Choosing points along the vector pointing to the current point is a one-dimensional problem, so we need only examine the most extreme points from each class.

If we have only one kernel in our MKL problem, then we skip the exponentiation step (since it is redundant). The only step remaining is to consult **Algorithm 5**. **Algorithm 5**, however, is equivalent to the polytope distance algorithm for two polytopes [30], so the single-kernel version of our algorithm is equivalent to a kernelized version of Gilbert [31].

3.4 Experiments

In this section, we compare the empirical performance of MWUMKL with other MKL 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 uniformly weighted combination of kernels (UNIFORM) and LibLinear with Nyström kernel approximations (LIBLINEAR+) as baselines. 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* (see **Table 3.1**).

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.

3.4.1 Uniform Kernel Weights

UNIFORM is simply LibSVM [18] run with a kernel weighted equally amongst all of the input kernels (where the kernel weights are normalized by the trace of their respective

Table 3.1: Datasets used in experiments.

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

Gram matrices first). The performance of UNIFORM is on par or better than LIBLINEAR+ on many datasets (see **Figure 3.1**) 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 3.2**), because of the need to hold the entire Gram matrix in memory[5]. We employ Scikit-learn [64] because it offers efficient access to LibSVM.

3.4.2 LibLinear with Nyström Kernel Approximations

One important observation about MKL 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 [28]), and transform the features of the data with a kernel approximation. For our purposes, we use Nyström approximations as described by Williams and Seeger [76] and discussed further by Yang et al. [82]. Because LibLinear is a primal method, we do not need to scale each kernel – each kernel manifests as a set of features, which the algorithm weights by

[5]This is true even when LibSVM is told to use one kernel, which it can compute on the fly – the scaling of LibSVM is $O(n^2)$ - $O(n^3)$ [18], poor compared to MWUMKL and LIBLINEAR+ with increasing sample size.

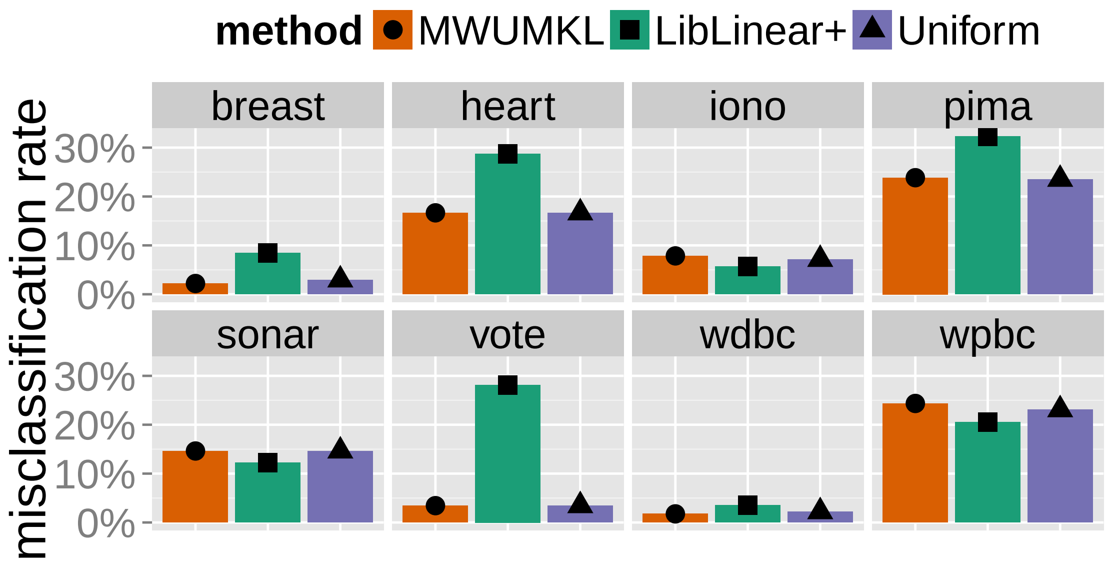

Figure 3.1: Median misclassification rate for small datasets.

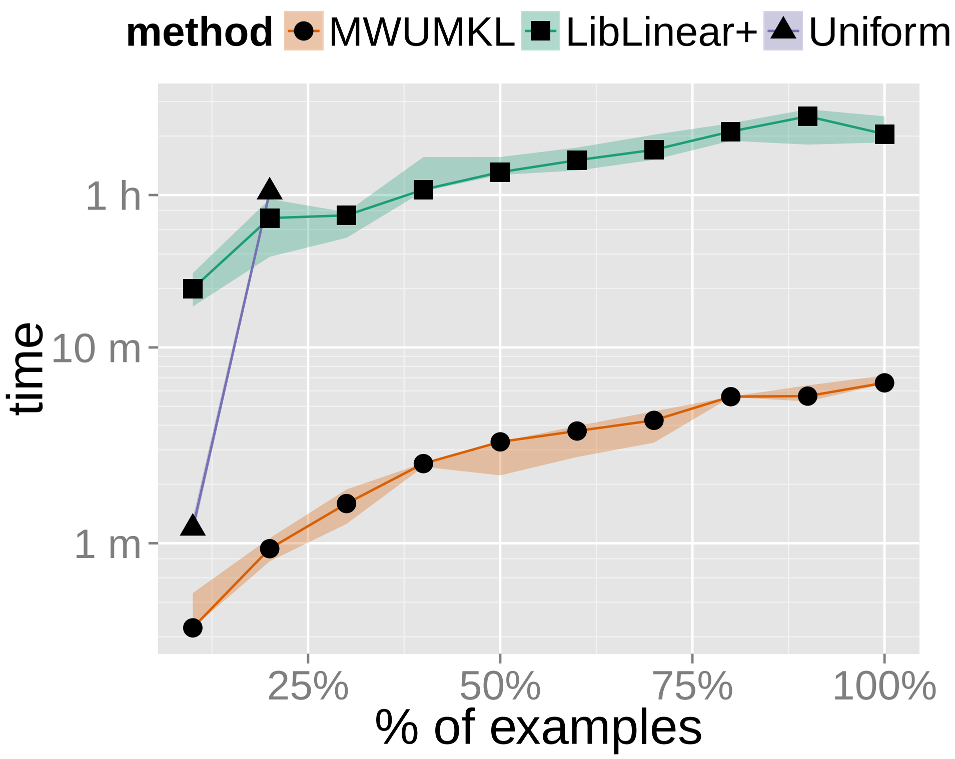

Figure 3.2: *CodRna* ($n = 59535$, $d = 8$) with 12 kernels.

definition.

For the Nyström feature transformations, one only needs to specify the kernel function and the number of sample points desired from the dataset. 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 [64] 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 **Figure 3.1**, **Figure 3.3**, and **Figure 3.4**). For high numbers

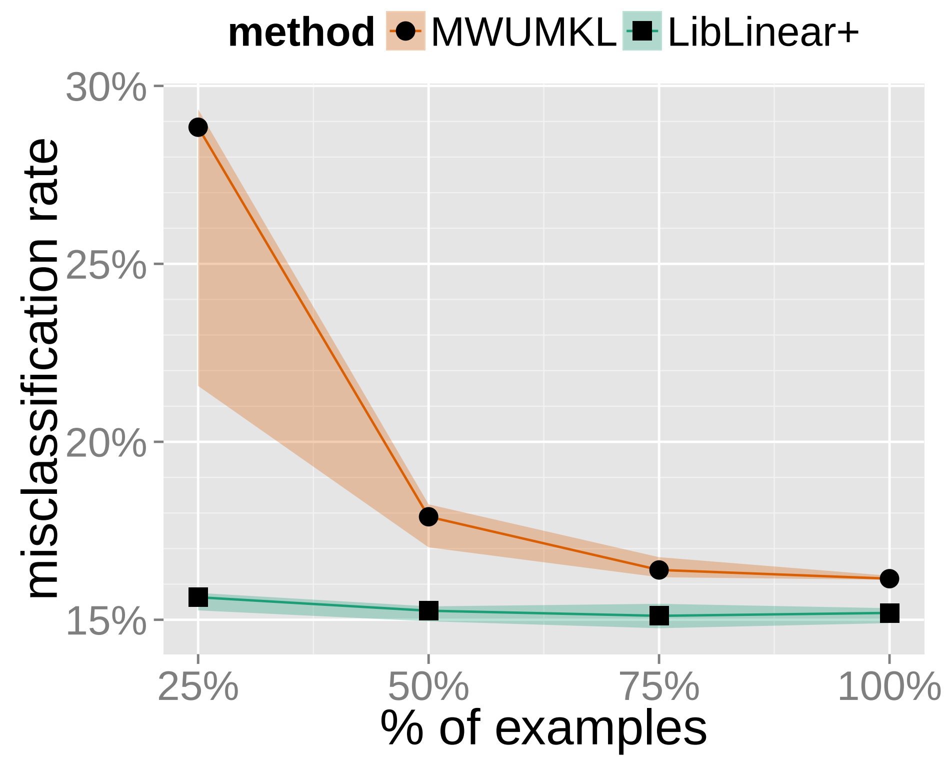

Figure 3.3: *Adult* ($n = 48842$, $d = 123$) with $m = 12$ kernels

of kernels, this scaling breaks down due to time and memory constraints (see **Figure 3.5**).

3.4.3 Legacy MKL Implementations

In all cases, we omit the results for older MKL algorithm implementations such as (a) SILP [69], (b) SDPMKL [49], (c) SIMPLEMKL [67], (d) LEVELMKL [79], and (e) GROUP-MKL [80] 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.

3.4.4 Experimental Parameters

Similar to Rakotomamonjy et al. [67] and Xu et al. [80], we test our algorithms on a base kernel family of 3 polynomial kernels (of degree 1 to 3) and 9 Gaussian kernels. Contrary to [67, 80], 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

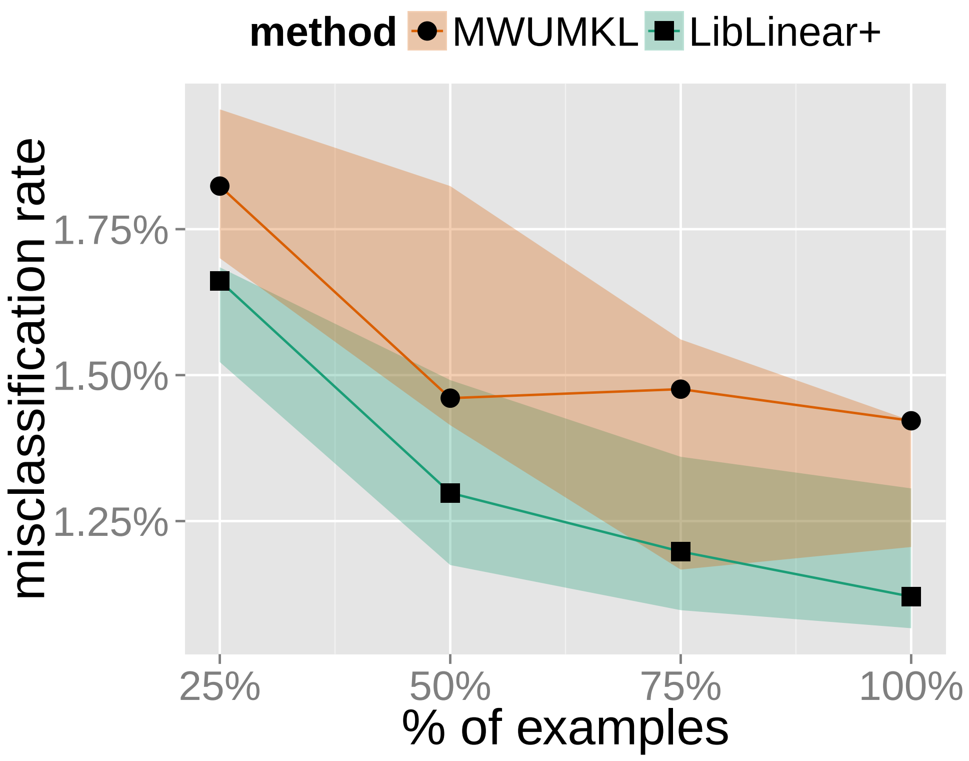

Figure 3.4: *Web* ($n = 64700$, $d = 300$) with $m = 12$ kernels

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 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 3.1**, results are presented for the best value of C for each dataset and algorithm.

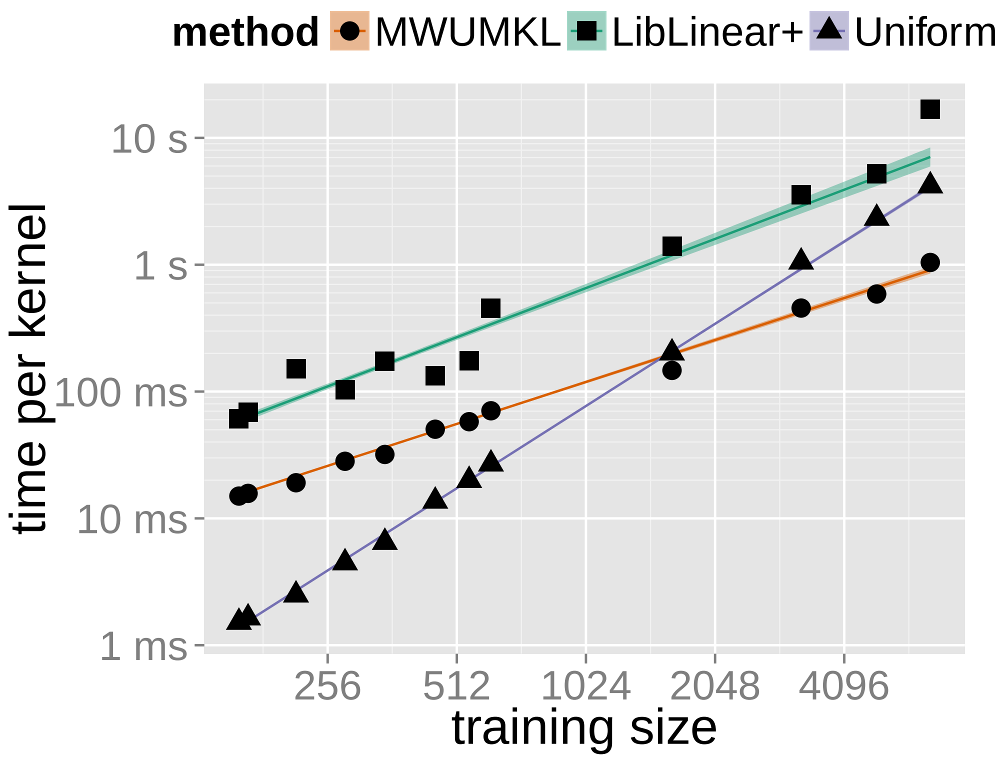

Figure 3.5: Time per kernel vs. data size for small and medium datasets (log-log).

For MWUMKL, we choose ε 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$^{\text{TM}}$ 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++.

3.4.5 Accuracy

On small datasets, our goal is to show that MWUMKL compares favorably with LibLinear with Nyström kernel approximations (LIBLINEAR+) and UNIFORM in terms of test accuracies.

In **Figure 3.1**, 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.

3.4.6 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 3.2**, 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, for the remaining experiments on large datasets, we compare MWUMKL with LibLinear+. In **Figure 3.3** and **Figure 3.4**, 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+.

3.4.7 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 3.5**). 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 3.1**, 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.

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 does not 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.

3.4.8 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 [75]). Table **Table 3.2** shows that we can indeed scale well beyond tens of thousands of points, as well as many kernels.

We choose the above datasets to compare against another work on scalable MKL [40]. Jain et al. [40] 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. [40] 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. [40].

Table 3.2: MWUMKL with on-the-fly kernel computations.

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

3.5 How to Shave a Factor of $1/\delta$ from Our Bound

This applies generally to any SDP where you can guarantee that $\mathbf{Q}(\mathbf{y}^{(t)}) = \sum_j \mathbf{A}_j y_j^{(t)} - \mathbf{C}$ has a tighter constraint on the eigenvalues.

3.5.1 Replacing Parts of the Algorithm

This section refers to Section 4.4 of Kale's PhD thesis [43]. The reader should follow along with that document.

3.5.1.1 The $(\ell, \rho)^*$-bounded ORACLE

We change Definition 2 slightly to accomodate our changes. We introduce a modified definition:

Definition 2. An $(\ell, \rho)^*$-bounded ORACLE, for parameters $\rho \geq 0$ and $|\ell| \leq \rho$, is an algorithm that finds a vector $\mathbf{y} \in \mathcal{D}_\alpha$ that satisfies Kale [43, Formula (4.1)] such that either $\mathbf{Q}(\mathbf{y}^{(t)}) \in [-\ell, \rho]$ or $\mathbf{Q}(\mathbf{y}^{(t)}) \in [-\rho, \ell]$ holds. The value ρ is called the *width* of ORACLE.

We assume that we replace a (ℓ, ρ)-bounded ORACLE with a $(\ell, \rho)^*$-bounded ORACLE. The only difference between the two is that a $(\ell, \rho)^*$-bounded ORACLE is allowed to let $\ell < 0$ (i.e., we can force $\mathbf{Q}(\mathbf{y}^{(t)}) \succeq 0$ and analyze that case).

3.5.1.2 ε and T

We set $\varepsilon = \frac{1}{2}$ and $T = \frac{2(\ell+\rho)R\ln n}{\delta\alpha}$. These changes shave the factor of $\frac{1}{\delta}$. The change to ε also eliminates the need for a lower bound on ℓ, and in fact opens up the other part of the range $(-\rho, \frac{\delta\alpha}{R}]$. ℓ was constrained to $[\frac{\delta\alpha}{R}, \rho]$ if $\varepsilon = \delta\alpha/2\ell R$, then $\varepsilon \leq \frac{1}{2}$, which is required by the Matrix Multiplicative Weight Updates algorithm. Here we satisfy the bound on ε by fiat, and as it turns out, gives us the other part of the range for ℓ.

3.5.1.3 $\mathbf{M}^{(t)}$

The purpose of setting $\mathbf{M}^{(t)}$ equal to $\frac{1}{\ell+\rho}[\mathbf{Q}(\mathbf{y}^{(t)}) + \ell^{(t)}\mathbf{I}]$ in Kale [43, Formula (4.2)] is to guarantee the positive (negative) semidefiniteness of $\mathbf{M}^{(t)}$. Either $\mathbf{M}^{(t)} \in [0,1]$ or $\mathbf{M}^{(t)} \in [-1,0]$, so it may be used in Kale [43, Formula (3.3)]. In the case of a $(\ell, \rho)^*$-bounded ORACLE, however, allowing $\ell < 0$ lets us *reduce* the eigenvalues. Note that if $\ell < 0$, we do not need to branch $\ell^{(t)} = \pm\ell$. If we can ever guarantee that $\mathbf{Q}(\mathbf{y}^{(t)}) \preceq 0$, then

$\mathbf{Q}(\mathbf{y}^{(t)}) \bullet \mathbf{X}^{(t)} \leq 0$, and ORACLE will reject $\mathbf{y}^{(t)}$.

3.5.2 Proof of Kale [43, Theorem 13]

The proof of Theorem 13 goes through with some minor changes. $\mathbf{M}^{(t)}$ is the same, so $\mathbf{M}^{(t)} \bullet \mathbf{P}^{(t)}$ is the same as well:

$$\mathbf{M}^{(t)} \bullet \mathbf{P}^{(t)} = \frac{1}{\ell + \rho} \left[\mathbf{Q}(\mathbf{y}^{(t)}) + \ell^{(t)}\mathbf{I}\right] \bullet \frac{1}{R}\mathbf{X}^{(t)} \geq \frac{\ell^{(t)}}{\ell + \rho}.$$

Plugging this into Kale [43, Formula (3.3)], we still get that

$$\begin{aligned} 0 &\leq \lambda_n\left(\mathbf{Q}(\bar{\mathbf{y}})\right) + \varepsilon\ell + \frac{(\ell+\rho)\ln n}{\varepsilon T} \\ -\frac{\delta\alpha}{R} &\leq \lambda_n\left(\mathbf{Q}(\bar{\mathbf{y}})\right) \\ -\frac{\delta\alpha}{R}\mathbf{I} &\preceq \mathbf{Q}(\bar{\mathbf{y}}), \end{aligned}$$

as long as

$$\varepsilon\ell + \frac{(\ell+\rho)\ln n}{\varepsilon T} \leq \frac{\delta\alpha}{R}. \tag{3.6}$$

For the case in [43], ε varies with δ, and ℓ is lower-bounded by $\frac{\delta\alpha}{R}$ so that $\varepsilon \leq \frac{1}{2}$. If we transform Inequality (3.6) so that T is alone on the left, we get:

$$\begin{aligned} \varepsilon\ell + \frac{(\ell+\rho)\ln n}{\varepsilon T} &\leq \frac{\delta\alpha}{R} \\ \frac{(\ell+\rho)\ln n}{\varepsilon T} &\leq \frac{\delta\alpha}{R} - \varepsilon\ell = \frac{\delta\alpha - \varepsilon\ell R}{R} \\ \frac{\varepsilon T}{(\ell+\rho)\ln n} &\geq \frac{R}{\delta\alpha - \varepsilon\ell R} \\ T &\geq \frac{1}{\varepsilon}\frac{(\ell+\rho)R\ln n}{\delta\alpha - \varepsilon\ell R}. \end{aligned}$$

If ℓ is large, it is easy to see that we would need to construct ε in such a way that the denominator remains positive. The most efficient way to do this is to choose ε so that $\varepsilon\ell R = \frac{\delta\alpha}{2}$. Unfortunately, this also means that T is quadratic in $\frac{1}{\delta}$.

For $\ell \in \left(0, \frac{\delta\alpha}{R}\right]$, we can shuffle things around, set ε to be a constant $\frac{1}{2}$, and get T to scale linearly in $\frac{1}{\delta}$, but since the point is to scale T against δ, this is irrelevant. As δ gets smaller, once $\frac{\delta\alpha}{R}$ drops below ℓ, T scales quadratically with $\frac{1}{\delta}$.

For negative ℓ, we can simply toss out the $\varepsilon\ell$ term:

$$\begin{aligned}\varepsilon\ell + \frac{(\ell+\rho)\ln n}{\varepsilon T} \leq \frac{(\ell+\rho)\ln n}{\varepsilon T} &\leq \frac{\delta\alpha}{R} \\ T &\geq \frac{1}{\varepsilon}\frac{(\ell+\rho)R\ln n}{\delta\alpha}.\end{aligned}$$

Setting an aggressive $\varepsilon = \frac{1}{2}$ gives us a bound linear in $\frac{1}{\delta}$. We can now build a complete description of what values to give ε and T for any value of ℓ:

$$\begin{array}{lll} \ell \leq 0: & \varepsilon = \dfrac{1}{2} & T = \dfrac{2(\ell+\rho)R\ln n}{\delta\alpha} \\ \ell > 0: & \varepsilon = \dfrac{\delta\alpha}{2\ell R} & T = \dfrac{4\ell(\ell+\rho)R^2\ln n}{\delta^2\alpha^2} \; \left(T = \dfrac{8\ell\rho R^2\ln n}{\delta^2\alpha^2}\right) \end{array}$$

CHAPTER 4

LOCALIZED DECISION-BASED MULTIPLE KERNEL LEARNING

Most multiple kernel learning (MKL) methods seek the combined kernel that performs best over *every* training example, sacrificing performance in some areas to seek a global optimum. *Localized* kernel learning (LKL) overcomes this limitation by allowing the training algorithm to match a component kernel to the examples that can exploit it best. Several approaches to the localized kernel learning problem have been explored in the last several years. We unify many of these approaches under one simple system and design a new algorithm with improved performance. We also develop enhanced versions of existing algorithms, with an eye on scalability and performance.

4.1 Introduction

While MKL has been studied extensively and has had success in identifying the right kernel for a given task, it is expressively limited because each kernel has influence over the entire data space. Consider an example of a binary classification task, depicted in **Figure 4.1**. On the left side, we show the results of classifying the data with a global MKL method (here, the UNIFORM method of Cortes et al. [25]) and on the right side, we show the results of classification with our new proposed method LD-MKL. Because the global method requires that each kernel be used to classify each point in the same way, the decision boundary is not as flexible and many more support points are required.

Motivated by this, a few directions have been proposed to build *localized* kernel learning solutions. Gönen and Alpaydin [34] introduced the idea of a learned *gating* function that modulated the influence of a kernel on a point (LMKL). Lei et al. [52] observed that LMKL uses a non-convex optimization and suggested using a probabilistic clustering to generate part of the gating function beforehand, in order to obtain a convex optimization and thus prevent over-fitting and yield generalization bounds (C-LMKL). Kannao and

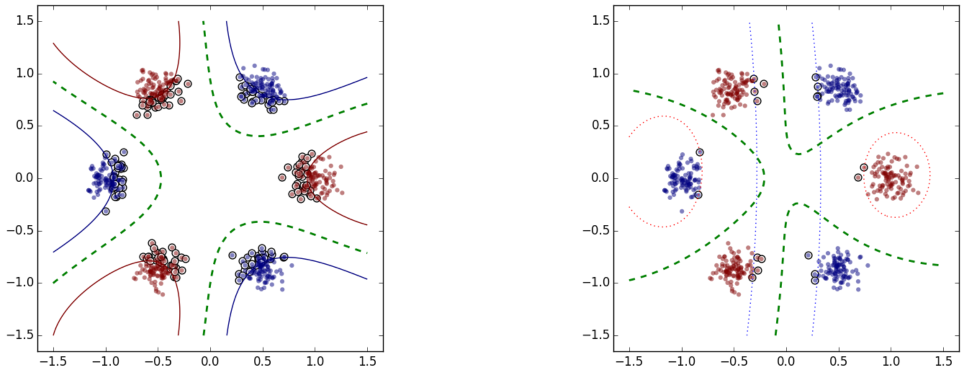

Figure 4.1: Illustration of the difference between global (left) and local (right) multiple kernel learning. In each example, the classifier is built from two kernels, one quadratic and one Gaussian. Points from the two classes are colored blue and red (with transparency as a hint towards density). The decision boundary is marked in green and the margin boundaries are in the appropriate colors for the global case. For the local case, the margins of each kernel are plotted with dotted lines, red for Gaussian and blue for quadratic. Support points are indicated by black circles around points. Note that the classifier uses a soft-margin loss and so support points may not be exactly on the margin boundary. The global version (left) has 118 support points, while the local version (right) has only 20.

Guha [44] suggested a different approach to find a gating function by looking at individual features of the input, and uses successes of the individual kernels to learn the gating function through support vector regression (SwMKL).

All of the above approaches invoke a fixed-kernel support vector machine (SVM) subroutine as part of the algorithm. This is inefficient, and prevents these methods from scaling. C-LMKL does argue for a convex formulation of the problem, but does not directly address the problem of scaling.

4.1.1 Our Contributions

We present a *unified* interpretation of localized kernel learning that generalizes all of the approaches described above, as well as the general MKL formulation. This interpretation yields a new algorithm for LKL that is superior to all existing methods. In addition, we make use of prior work on scalable MKL (Chapter 3) as a subroutine to make existing methods for LKL scale well, improving their performance significantly in some cases.

Our interpretation relies on a geometric interpretation of gating functions in terms of *local* reproducing kernel Hilbert spaces acting on the data. This interpretation also helps explain the observation above (only empirically observed thus far) that local kernel learning methods appear to produce good classifiers with fewer support points than global methods.

4.2 Background

Because we discuss several approaches to localized MKL, and each uses a different set of notations, we choose our own convention:

- i indexes kernel functions/spaces and the number of individual kernel spaces is m.
- j and k index examples and the number of training points is n.
- t is used to indicate iterations in an algorithm.
- The Greek letter κ is used to indicate a kernel function. $\kappa_i(\mathbf{x}_j, \mathbf{x}_k)$ is the ith kernel function applied to training examples $\mathbf{x}_j$ and $\mathbf{x}_k$.

4.3 A Unified View of Localized Kernel Learning

One of the contributions of this work is a unified perspective that integrates these different approaches and also helps explain the somewhat paradoxical fact that localized MKL often yields classifiers with *fewer* support points than standard MKL methods.

4.3.1 Localization via Hilbert Subspaces

Consider the following generalized and gated kernel κ_γ defined as:

$$\kappa_\gamma(\mathbf{x}, \mathbf{x}') = \sum_{i=1}^{m} \gamma_i(\mathbf{x}, \mathbf{x}')\kappa_i(\mathbf{x}, \mathbf{x}'),$$

where $\gamma_i : \mathbb{R}^d \times \mathbb{R}^d \to [0, 1]$ is a "gating function."

We call γ_i *separable* if it decomposes into a product of a function with itself, i.e., if $\gamma_i(\mathbf{x}, \mathbf{x}') = \eta_i(\mathbf{x})\eta_i(\mathbf{x}')$, where $\eta_i : \mathbb{R}^d \to [0, 1]$. For the rest of this section, we only consider separable gating functions. We also make two additional assumptions for all $\mathbf{x} \in \mathbb{R}^d$: (1) $\sum_{i=1}^{m} \eta_i(\mathbf{x}) = 1$, and (2) $\eta_i(\mathbf{x}) \geq 0 \; \forall i \in [1..m]$.

4.3.1.1 The RKHS of a Localized Kernel

Consider the Gram matrix $\mathbf{H}_i$ of γ_i: specifically the $n \times n$ matrix $\mathbf{H}_i$ whose $(j, k)^{\text{th}}$ entry is $\gamma_i(\mathbf{x}_j, \mathbf{x}_k)$ (we will refer to this later as the *gating matrix*). If γ_i is separable, then we know that $\mathbf{H}_i$ is positive definite, because it can be expressed as the outer product of a vector with itself ($\mathbf{H}_i = \boldsymbol{\eta}^\top \boldsymbol{\eta}$). Defining $\mathbf{K}_i$ as the Gram matrix of the kernel κ_i, it is now easy to see that we can write the Gram matrix of the kernel κ_γ as the matrix $\sum_i \mathbf{H}_i \circ \mathbf{K}_i$.

In the separable case, since both $\mathbf{H}_i$ and $\mathbf{K}_i$ are positive definite, so is $\mathbf{H}_i \circ \mathbf{K}_i$ by the Schur product theorem. Therefore, $\gamma_i(\mathbf{x}, \mathbf{x}')\kappa_i(\mathbf{x}, \mathbf{x}')$ is a positive-definite (p.d.) kernel, and the corresponding lifting map is $\eta_i(\mathbf{x})\Phi_i(\mathbf{x})$.

We know that a positive linear combination of kernel functions is itself a kernel function and induces a product reproducing kernel Hilbert space (RKHS) that is a simple Cartesian product of all the individual Hilbert spaces. The inner product of this space is just the sum of all the individual inner products. Thus the kernel κ_γ has a natural feature space as the product of the individual feature spaces.

4.3.1.2 Localization

This framework now allows us to provide a geometric intuition for why localized kernel learning might be able to reduce the number of required support points. Suppose that $\eta_i(\mathbf{x}) = 0$. This implies that $\langle \eta_i(\mathbf{x})\Phi_i(\mathbf{x}), \eta_i(\mathbf{x}')\Phi_i(\mathbf{x}') \rangle$ is always 0. Because the i^{th} RKHS is one component of the product RKHS, this means that $\eta_i(\mathbf{x})\Phi_i(\mathbf{x})$ lies in some subspace perpendicular to this RKHS.

Furthermore, suppose that $\eta_i(\mathbf{x}) = 1$. By our assumptions that $\sum_{i=1}^{m} \eta_i(\mathbf{x}) = 1$ and that η_i is non-negative, this means that $\eta_i(\mathbf{x})\Phi_i(\mathbf{x})$ is absent from *every other* RKHS in the product. Therefore, $\eta_i(\mathbf{x})\Phi_i(\mathbf{x})$ lies exclusively in the i-th RKHS.

This partitioning behavior is advantageous, because it is much simpler to find decision boundaries within the individual RKHS components rather than trying to find one that will work for all at the same time. The decision hyperplane in the product RKHS will be the unique hyperplane that intersects all the subspaces in their respective decision boundaries.

Depending on the gating function, there will of course be some training examples that are “confused” about what subspace to lie in. Therefore, we wish to pick a set of gating functions that reduces this confusion. The *crucial* property of the gating function γ_i and the gating matrix $\mathbf{H}_i$ is that they are separable. With the separability constraint, we need only find a set of one-dimensional functions that works for the training data[1].

[1] If the gating function is not separable, but is decomposable into a positive linear combination of a fixed-

4.3.2 Gating and Optimization

The localized MKL algorithms described above (and in fact virtually all localized kernel learning algorithms) can be placed in the framework we have just described, thus explaining in a broader context how their localization works. The specifics differ on how the function κ_γ is generated:

1. **Gating**: Each algorithm has a gating function $\gamma_i(\mathbf{x}, \mathbf{x}')$ for every kernel function κ_i. Recall that the gating function simply controls the degree to which a kernel responds to a particular point.

2. **Optimization**: Each algorithm also has an optimization behavior that either generates or tunes each γ_i.

4.3.2.1 LMKL

- **Gating**: The gating function is separable, and $\eta(\mathbf{x}) = \mathrm{softmax}(\mathbf{x}^\top \mathbf{V} + \mathbf{v}_0)$.

- **Optimization**: Alternating optimization using an SVM solver to find the kernel support points and stochastic gradient descent to find the parameters $\mathbf{V}$, $\mathbf{v}_0$.

4.3.2.2 C-LMKL

- **Gating**: The gating function is separable, but not directly. $\eta(\mathbf{x}) = \sum_{r=1}^{\ell} \beta_{ir} c_r(\mathbf{x}) c_r(\mathbf{x}')$, where $\beta_{ir} \geq 0$ is the weight with which kernel i influences points associated with cluster r, and c_r is the (precomputed) likelihood of $\mathbf{x}$ falling into cluster r.

  Since γ_i decomposes into a linear combination $\beta_{ir} c_r(\mathbf{x}) c_r(\mathbf{x}')$, we can apply Section 4.3.1 to C-LMKL. In C-LMKL, we replicate each kernel ℓ times (once for each c_r) and give each its own weight $\sqrt{\beta_{ir}}$.

- **Optimization**: The parameters β_{ir} are learned through (convex) optimization and the functions c_r are generated through ℓ different clusterings.

size set of separable functions, then the partitioning is still possible – see Section 4.3.2.2 below, under "C-LMKL".

4.3.2.3 SwMKL

- **Gating**: The gating function is not separable in this case, because the γ_i are normalized *pairwise*. $\gamma_i(\mathbf{x}, \mathbf{x}') = g_i(\mathbf{x})g_i(\mathbf{x}')/Z(\mathbf{x}, \mathbf{x}')$, where $Z(\mathbf{x}, \mathbf{x}') = \sum_{i=1}^{m} g_i(\mathbf{x})g_i(\mathbf{x}')$, and g_i are the SVR-generated functions.

  Note that while κ_γ may be positive definite, its individual terms are very unlikely to be so. It is therefore not clear whether this algorithm in its unmodified form can be placed in our unified context. We explore this issue in greater depth in the next section.

- **Optimization**: The gating functions g_i are generated using SVR from $\mathbf{X} \times \delta(\mathbf{y}, \hat{\mathbf{y}}_i)$.

4.3.2.4 SAMKL

- **Gating:** $\eta_i(\mathbf{x})$ is a *binary-valued* function that decides if kernel i should be used for point $\mathbf{x}$.

- **Optimization:** The optimization is an alternating optimization between the gating function and the kernel parameters. Because the η_i are binary-valued, a further *multiple* kernel learning step is required to determine kernel weights and support vectors for the classifier, and the gating parameters are learned with an *integer programming* solver.

4.3.2.5 Global ("classic") MKL

- **Gating**: $\eta_i(\mathbf{x}) = \sqrt{\mu_i}$, where $\mu_i \geq 0$ is constant for every kernel, that is, does not change relative to each point.

- **Optimization**: The μ_i can be optimized using several methods including stochastic gradient descent, multiplicative weight updates, and alternation.

4.4 LD-MKL: A New Algorithm for Localized Kernel Learning

Viewing the algorithms for localized kernel learning in a common framework illustrates both their commonalities and their weaknesses. With the exception of SwMKL, all the approaches make use of a two- (or three-) stage optimization of which `LibSVM` is one component. As we shall see in our experiments, this renders these methods quite slow and not easy to scale. SwMKL on the other hand avoids this problem by doing single

SVM calculations for each kernel and then combining them into a single larger kernel. This improves its running time, but makes it incur a large memory footprint in order to build a classifier for the final kernel.

We now present a new approach, inspired by SWMKL, that addresses these concerns. Our method, which we call LD-MKL (*localized decision-based MKL*), fits into the unified framework for localized kernel learning via the use of local Hilbert spaces, avoids the large memory footprint of SWMKL, and also scales far more efficiently than the other multistage optimizations.

We start by observing that the first steps of **Algorithm 2** give us a classifier f_i and a gating function g_i. The function f_i, since it is an SVM decision function, can be formulated as

$$f_i(\mathbf{x}) = \sum_{j=1}^{n} \alpha_{ij} y_j \kappa_i(\mathbf{x}_j, \mathbf{x}).$$

Note that α has an additional index to indicate which kernel we trained the classifier against. Suppose we modify this function to incorporate the gating function g_i[2]:

$$\overline{f}_i(\mathbf{x}) = \sum_{j=1}^{n} \alpha_{ij} y_j g_i(\mathbf{x}_j) \kappa_i(\mathbf{x}_j, \mathbf{x}). \tag{4.1}$$

$\overline{f}_i$ is the SVM prediction function, but where each support point α_{ij} is weighted by its gating value. We can now construct a weighted vote using these functions. We combine the output of each $\overline{f}_i$, apply tanh[3], and weight by g_i:

$$f(\mathbf{x}) = \sum_{i=1}^{m} g_i(\mathbf{x}) \tanh(\overline{f}_i(\mathbf{x})) \tag{4.2}$$

Algorithm 7 contains the listing of this procedure. Note that we *retrain* each classifier on the subset of the data where the corresponding gating function is significant (i.e., is greater than $1/m$). This reduces the support points considerably because the classifier is retrained only on points that it classified well.

If commonly-used kernels are employed (such as linear, polynomial, or Gaussian kernels), then this method can take advantage of optimizations that exist in, e.g., LibSVM to

[2]As discussed in the previous section, we assume that the gating functions have been normalized so that (1) $\sum_{i=1}^{m} g_i(\mathbf{x}) = 1$ and (2) $g_i(\mathbf{x}) \geq 0 \; \forall i \in [1..m]$.

[3]We use $\tanh(\overline{f}_i(\mathbf{x}))$ instead of the sign of $\overline{f}_i(\mathbf{x})$ so that uncertain classifications (i.e., kernels with resulting values of $\overline{f}_i(\mathbf{x})$ near 0) do not pollute the vote with noise.

Algorithm 7 LD-MKL

```
1: for all $i \in [1..m]$ do
2:     Train classifier $f_i : \mathbb{R}^d \to \{-1, 1\}$ with kernel $\kappa_i$
3:     Train regressor $g_i : \mathbb{R}^d \to (0, 1)$ with $(\mathbf{X}, \delta(\mathbf{y}, f_i(\mathbf{X})))$
4: Normalize regressors $g_i$ with softmax
5: for all $i \in [1..m]$ do
6:     Retrain classifier $f_i$ on $(\mathbf{X}, \mathbf{y})_{g_i(\mathbf{x})>1/m}$
7: Compute each decision function using (4.1)
8: Classify inputs using sign of (4.2)
```

train the classifiers and regressors quickly. The training step is over after the regressors are computed and normalized.

It is easy to see that LD-MKL has the desired *gating* behavior with separable gating functions. The optimization step is as before, but without needing to consult a final SVM solver.

4.5 Experiments

Our experiments will seek to validate two main claims: first that LD-MKL is indeed superior to prior localized kernel learning methods, and second that there is demonstrable reduction in the number of support points when using localized methods.

4.5.1 Scalability

In addition, we will also investigate ways to make existing localized methods more scalable. As noted, with the exception of SWMKL, all approaches use a multistage iterative optimizer of which one step is an SVM solver. We instead make use of MWUMKL, described in Chapter 3. This method has a much smaller memory footprint and uses a lightweight iteration that also yields sparse support vectors. While this solver was designed for *multiple* kernel learning, it is easily adapted as an SVM solver.

4.5.2 Datasets

Table 4.1 contains information about the various datasets that we test with. All of these sets are taken from the `libsvm` repository[4].

[4]`https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary.html`

Table 4.1: Datasets for comparison of LMKL, SWMKL, and C-LMKL

Dataset	Examples	Features
Breast Cancer	683	10
Diabetes	768	8
German-Numeric	1000	24
Liver	345	6
Mushroom	8192	112
Gisette	6000	5000
Adult	32561	123

4.5.3 Methodology

In each of the experiments, we partition the data randomly between 75% train and 25% test examples. Unless otherwise indicated, we repeat each partition 100 times and average the run time and the accuracy. In all experiments where we measure accuracy, we use the proportion of correctly classified points. Where possible, we also report the standard deviation of all measured values in parentheses. Superior values are presented in bold when the value minus the standard deviation is greater than all the other values plus their respective standard deviations.

In each experiment where we used a standard SVM solver, we used LibSVM [18] via `scikit-learn` [64]. We use the default LibSVM parameters (e.g., tolerance), and vary them only for changing specific kernels and passing specific kernel parameters. We use $C = 1.0$ and for Gaussian kernels, a range of γ from 2^{-4} to 2^{4} are tried and the best accuracy observed is used.

4.5.4 Implementations

For LMKL, we took MATLAB code provided by Gönen and Alpaydin [34][5] and converted it to `python` to have a common platform for comparison. This code included an SMO-based SVM solver which we converted as well. We verified correctness of intermediate and final results between the two platforms before running our experiments. For SWMKL and LD-MKL, we used the SVM and SVR solvers from `scikit-learn`. For C-LMKL, as prescribed by Lei et al. [52], we used a kernel k-means preprocessing step with a uniform kernel and three clusters. For large datasets, kernel k-means is very slow,

[5]`http://users.ics.aalto.fi/gonen/icml08.php`

and so we used a streaming method proposed by Chitta et al. [20] that runs the clustering algorithm on a sample (of size 1000 in our experiments) and then estimates probabilities for the remaining points. The *global* kernel learning methods we used were UNIFORM, which merely averages all kernels, SPG-GMKL [40][6] and MWUMKL (Chapter 3). All experiments were conducted on Intel® Xeon® E5-2650 v2 CPUs, 2.60GHz with 64GB RAM and 8 cores.

4.5.5 Evaluating LD-MKL

We start with an evaluation of LD-MKL in **Table 4.2**. In each row, we present accuracy and timing (numbers in parentheses are standard deviations). As we can see, for small datasets, SWMKL is the fastest method, but for larger datasets, LD-MKL is the fastest. In comparison with LMKL and C-LMKL, SWMKL and LD-MKL are considerably faster. This speedup is obtained without any significant loss in accuracy: in all cases, the accuracy of LD-MKL is either the best or is less than optimal in a statistically insignificant way.

[6]`http://www.cs.cornell.edu/~ashesh/pubs/code/SPG-GMKL/download.html`

Table 4.2: Accuracies and running times for various datasets and methods, using **LibSVM** as the SVM solver. Numbers in parentheses are standard deviations. For the first four datasets, numbers are averaged over 100 runs. For the last three larger datasets, numbers are averaged over 20 runs. Values which are significantly superior to that of other methods are typeset in bold.

	LMKL	SWMKL	LD-MKL	C-LMKL
Breast	96.58 % (1.35 %)	97.1% (1.1%)	97.1% (1.2%)	96.7% (1.1%)
Cancer	122 s (8.9 s)	0.15 s (2.1 ms)	**0.14 s** (2.36 ms)	28.7 s (80 ms)
Diabetes	74.71% (3.07%)	77.0% (2.7%)	76.7% (2.56%)	76.4% (2.4%)
	157.6 s (34 s)	**0.18 s** (1.4 ms)	0.24 s (3.58 ms)	36.8 s (32 ms)
German-	70.78% (2.85%)	75.7% (2.4%)	75.84% (2.51%)	76.8% (1.6%)
Numeric	216 s (22 s)	**0.27 s** (3.8 ms)	0.38 s (3.56 ms)	69.6 s (20 ms)
Liver	62.49% (5.92%)	69.3% (5.0%)	65.19% (5.81%)	57.7% (5.1%)
	35.4 s (7.2 s)	**0.1 s** (1.4 ms)	0.83 s (1.3 ms)	7.4 s (129 ms)
Mushroom	99.99% (0.0%)	99.9% (0%)	100.0% (0.0%)	100% (0.0%)
	17.27 m (1.2 m)	14.57 s (1.2 s)	**3.1 s** (0.3 s)	2.43 h (12.6 m)
Gisette	97.22% (0.34%)	97.06% (0.35%)	96.88% (0.46%)	96.5% (0.28%)
	48.6 m (2.8 m)	4.54 m (0.72 m)	4.0 m (0.06 m)	3.4 h (9.24 m)
Adult	-	84.6% (0.37%)	84.78% (0.4%)	84.65% (0.83%)
Income	-	6.65 m (1.2 m)	6.52 m (0.14 m)	7.5 h (14.3 m)

4.5.6 Scaling

As we can see in **Table 4.2**, LMKL and C-LMKL run very slowly as the data complexity increases (dimensions or number of points), and the primary bottleneck is the repeated invocation of an SVM solver. As described above, we replaced the SVM solver with a single-kernel version of MWUMKL and studied the resulting performance.

Table 4.3 summarizes the results of this experiment. As we can see, for both LMKL and C-LMKL, using a scalable SVM solver greatly improves the running time of the algorithm. In fact as we can see, the methods using LibSVM fail to complete on certain inputs, whereas the methods that use MWUMKL do not. We note that MWUMKL uses a parameter ϵ which is the acceptable error in the duality gap of the SVM optimization program. Higher ϵ values translate to more iterations, and accuracy can often improve (up to a point) with lower ϵ. Unless stated otherwise, we use $\epsilon = 0.01$. Note that for this ϵ, accuracy does drop significantly in certain cases.

The case of SWMKL is a little more interesting. For smaller datasets the basic method works quite well, and indeed outperforms any enhancement based on using MWUMKL. However, this comes at a price: the SWMKL method requires a lot of memory to solve the

Table 4.3: Accuracies and running times for various datasets and methods, using MWUMKL as the SVM solver. Numbers in parentheses are standard deviations. For the first four datasets, numbers are averaged over 100 runs. For the last three larger datasets, numbers are averaged over 20 runs. Values which are significantly superior to that of other methods are typeset in bold.

	LMKL	SWMKL	LD-MKL	C-LMKL
Breast	97.08 % (1.1 %)	96.42% (1.6%)	93.4% (2.2%)	90.4% (2.4%)
Cancer	0.18 s (4.3 ms)	0.18 s (1.2 ms)	0.63 s (9.3 ms)	5.6 s (122 ms)
Diabetes	73.19% (3.39%)	76.63% (2.9%)	77.0% (3.4%)	71.1% (10%)
	0.27 s (18 ms)	0.29 s (3.8 ms)	0.48 s (46 ms)	7.2 s (32 ms)
German-	70.07% (3.1%)	72.2% (3.29%)	73.0% (3.8%)	73.4% (4.1%)
Numeric	0.63 s (43 ms)	0.62 s (10.2 ms)	1.0 s (55 ms)	16.3 s (101 ms)
Liver	56.82% (6.53%)	59.63% (10.47%)	58.8% (8.0%)	49.7% (6.3%)
	0.13 s (5 ms)	**0.11 s** (3.4 ms)	0.3 s (6.5 ms)	1.45 s (106 ms)
Mushroom	99.87% (0.1%)	99.9% (0%)	99.9% (0.1%)	98.8% (0.24%)
	24.4 s (0.36 s)	**21.3 s** (0.2 s)	53.0 s (0.2 s)	31.4 m (1.2 m)
Gisette	**97.28%** (0.4%)	69.96% (2.01%)	92.2% (0.8%)	90.26% (1.2%)
	8.2 m (0.18 m)	8.91 m (0.44 m)	29.0 m (10 s)	28.5 m (53.1 s)
Adult	57.4% (5.31%)	83.96% (0.61%)	80.2% (0.8%)	84.65% (0.35%)
Income	9.4 m (0.6 m)	9.1 m (6.8 s)	12.3 m (14.3 s)	47.65 m (2.46 m)

final kernel SVM with a kernel formed by combining the base kernels. For smaller datasets, this effect does not materially affect performance, but as we move to larger datasets like Adult, the method starts to fail catastrophically. **Figure 4.2** illustrates the memory usage incurred by the three localized methods when not using MWUMKL and when using it. As we can see, the memory grows polynomially with the size of input.

4.5.6.1 Stress-testing

Scaling LD-MKL to truly large datasets can present a challenge because we make use of kernelized support-vector regression. There are several methods to address this problem which we will not enumerate here, but are targets for future versions of our algorithm.

4.5.7 Support Points

We have argued earlier that localized MKL methods have the potential to generate classifiers with comparable accuracy but fewer support points than global multiple kernel methods. This fact was first observed by Gönen and Alpaydin [33]. We now present detailed empirical evidence establishing this claim. We compare the different localized

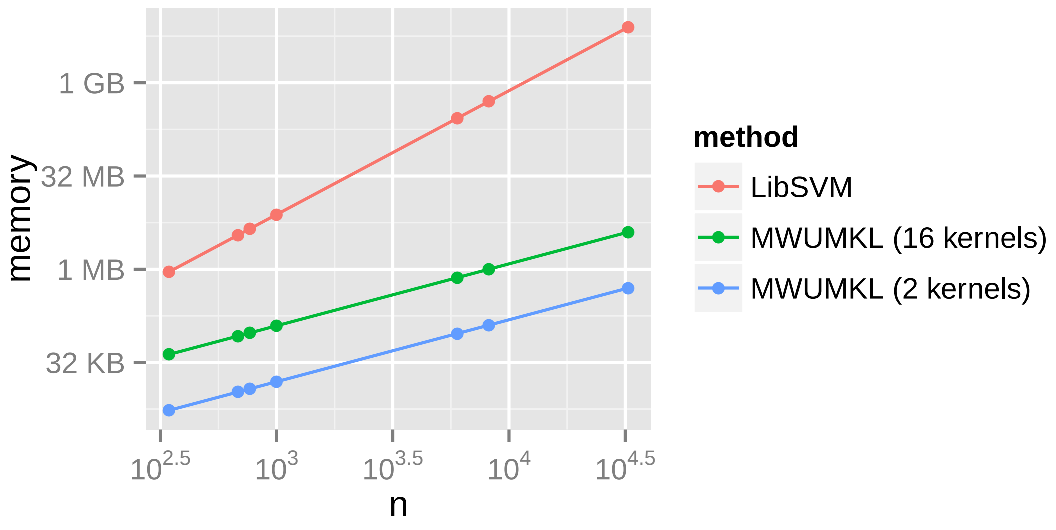

Figure 4.2: Minimum memory required (assuming double-precision floats) for LibSVM-based and MWUMKL-based methods. LibSVM-based methods exclude those that use only LibSVM's standard kernels, such as LD-MKL, but include those that construct a new kernel, such as LMKL, C-LMKL, and SwMKL. The values for n are taken from the "Examples" column from **Table 4.1**.

kernel learning methods to Uniform (a MKL algorithm that merely takes an average of all the kernels in its dictionary [23]), SPG-GMKL [40] (an iterative MKL solver that uses the spectral projected gradient), and MWUMKL, run in its original form as an MKL algorithm. Results are presented in **Table 4.4**. While we did not annotate the results with accuracy numbers for ease of viewing, all methods have comparable accuracy (as **Table 4.2** also indicates).

We observe that in all cases, the classifier using the fewest support points is always one of the localized methods, and the differences are always significant. However, it is not the case that a single local method always performs best. In general, LD-MKL (and SwMKL) appear to perform slightly better, but this is not consistent. Nevertheless, the results provide a clear justification for the argument that local kernel learning indeed finds sparser solutions.

Table 4.4: Numbers of support points computed as a percentage of the total number of points. Numbers in parentheses are standard deviations over 100 iterations. Values which are significantly superior to those of other methods are typeset in bold.

	Global methods			
	MWUMKL	Uniform	GMKL	
Breast	21% (1.4%)	70.2% (1.9%)	15.1% (1%)	
Diabetes	79.1% (1.7%)	70% (2%)	61.9% (1.2%)	
German	81.7% (1.1%)	60.8% (1.6%)	68.4% (1.4%)	
Liver	92.2% (1.6%)	89.6% (2.6%)	84.2% (1.9%)	
Mushrooms	22.6% (0.1%)	96.4% (0.8%)	15.2% (0.2%)	
Gisette	36.9% (0.0%)	99.4% (0.3%)	46.2% (0.3%)	
Adult	40.4% (0.0%)	48.2% (0.2%)	41.7% (0.1%)	
	Localized Methods			
	SwMKL	LD-MKL	LMKL	C-LMKL
Breast	11.4% (1%)	12.9% (1.1%)	38% (3.5%)	**10.8%** (1.1%)
Diabetes	**55.2%** (1.3%)	56.4% (1.3%)	58% (1.7%)	73.9% (10%)
German	52.2% (3.3%)	**43.4%** (2.4%)	89.2% (2.8%)	99.8% (0.3%)
Liver	82.2% (1.7%)	70.2% (7.3%)	**63.1%** (2.3%)	88.1% (2.7%)
Mushrooms	4.3% (0.2%)	8.1% (0.8%)	**1.9%** (0.1%)	4.0% (0.3%)
Gisette	**20.8%** (0.3%)	31.9% (0.2%)	32.3% (0.8%)	26.3% (0.5%)
Adult	35.6% (0.2%)	37.4% (0.2%)	-	35.4% (0.2%)

PART II

DISTRIBUTION-BASED KERNEL LEARNING

CHAPTER 5

CONTINUOUS KERNEL LEARNING

In this chapter, we describe a new approach to kernel learning that establishes connections between the Fourier-analytic representation of kernels arising out of Bochner's theorem [14] and a specific kind of feed-forward network using cosine activations. We analyze the complexity of this space of hypotheses and demonstrate empirically that our approach provides scalable kernel learning superior in quality to prior approaches.

5.1 Introduction

In this chapter, we describe *continuous kernel learning (CKL)*, a new way of tackling this problem by establishing and exploiting a connection to feed-forward networks. Working within the Fourier-analytic framework for kernel learning, we propose to search directly over the space of shift-invariant kernels instead of optimizing the parameters of a known family of distributions. In doing so, though we lose the ability to isolate parameters of a single learned kernel, we gain representability in terms of a nonlinear basis of cosines that can be naturally interpreted as activations for a feed-forward network. This interpretation allows us to deploy the power of backpropagation on this network to learn the desired kernel representation. In addition, the generalization power of the cosine representation can be established formally using machinery from learning theory: this also helps guide the regularization that we use to learn the resulting kernel. We support these arguments with a suite of experiments on relatively large datasets (tens of thousands of points, hundreds of dimensions) that demonstrate that our learned kernels are more accurate than the state-of-the-art multiple kernel learning (MKL) methods.

In summary, our main contributions are:

- We develop the CKL framework, a kernel learning method that learns an implicit representation of a kernel. We show that we can interpret the learning task as a feed-forward network. This allows us to utilize recent advances in optimization technology from deep learning to train a classifier.

- We prove VC-dimension and generalization bounds for a single Fourier embedding, which yields natural regularization techniques for CKL.

- We show via experiments that CKL outperforms existing scalable MKL methods.

5.1.1 Technical Overview

The starting point for our work is the representation of any shift-invariant kernel[1] as an infinite linear combination of cosine basis elements via Bochner's theorem [14], as first demonstrated by Rahimi and Recht [66]. This representation is typically used to generate a *random* low-dimensional embedding of the associated Hilbert space.

If we move away from a random low-dimensional embedding and embrace the entire distribution that we sample from, we reach infinite-width embeddings. Dealing with infinite-width embeddings simply means that we consider the expectation of the embedding over the distribution. Neal [59] linked infinite-width networks to Gaussian processes when the distribution is Gaussian. Much later, Cho and Saul [21] applied the technique to infinite-width *rectified linear units* (ReLUs), and showed a correspondence to a kernel they called the *arc-cosine kernel*. Hazan and Jaakkola [39] extended this result further, and analyzed the kernel corresponding to two infinite layers stacked in series. In all of this, a *specific distribution* is chosen in order to obtain a kernel.

In our work, we return to the infinite representation provided by Bochner's theorem [14]. Rather than picking a specific distribution over weights, we *learn* a distribution based on our training data. This effectively means we learn a *representation* of a kernel. While we cannot learn an infinite-width embedding directly, since the space of functions is itself infinite, we are able to construct approximate representations from a finite number of Fourier embeddings.

[1]A kernel $\kappa(x, y)$ expressible as $\kappa(x, y) = k(x - y)$.

5.2 Continuous Kernel Learning

5.2.1 Bochner's Theorem

A couple observations must be made in order for Bochner's theorem [14] to be relevant to our setting. First, we observe that (for the purposes of this chapter) a positive-definite (p.d.) *function* $k(\cdot)$ is a p.d. *kernel* $\kappa(\cdot,\cdot)$ when $\kappa(\mathbf{x},\mathbf{x}') = k(\mathbf{x}-\mathbf{x}')$ and k is even ($k(\delta) = k(-\delta)$). A kernel of this type is a *shift-invariant* kernel. Examples include the Gaussian or RBF kernel ($e^{-\|\mathbf{x}-\mathbf{x}'\|^2/\sigma^2}$) and the Laplacian kernel ($e^{-\lambda\|\mathbf{x}-\mathbf{x}'\|}$).

Next, any non-negative measure $\mu : \mathbb{R}^d \to \mathbb{R}^+$ can be converted to a probability distribution if we normalize by $Z = \int_{\mathbb{R}^d} d\mu$. Since Fourier transforms are linear, we can normalize the kernel by the same factor Z and maintain the equivalence. So without loss of generality, we can assume that the measure μ is a probability measure. This equivalence between shift-invariant kernel and distribution is important in the rest of this chapter.

5.2.2 Fourier Embeddings

Rahimi and Recht [66] built on Bochner's theorem [14] by observing that the Fourier transform of μ is also an expectation:

$$k(\mathbf{x}-\mathbf{x}') = \int_{\mathbb{R}^d} e^{i\boldsymbol{\omega}^\top(\mathbf{x}-\mathbf{x}')} f_\mu(\boldsymbol{\omega})\, d\boldsymbol{\omega} = E_\omega[\zeta_\omega(\mathbf{x})\overline{\zeta_\omega(\mathbf{x}')}],$$

if $\zeta_\omega(\mathbf{x}) = e^{i\boldsymbol{\omega}^\top\mathbf{x}}$ and $\boldsymbol{\omega} \sim \mathcal{D}_\mu$, where $\mathcal{D}_\mu$ is the probability distribution over Borel sets on $\mathbb{R}^d$ with measure μ. This shows that $\zeta_\omega(\mathbf{x})\overline{\zeta_\omega(\mathbf{x}')}$ is an unbiased estimate of $k(\mathbf{x}-\mathbf{x}')$. Because $k(\mathbf{x}-\mathbf{x}')$ is real, we know that $E_\omega[\zeta_\omega(\mathbf{x})\overline{\zeta_\omega(\mathbf{x}')}]$ has no imaginary component. A straight-forward Chernoff-type argument [see 58, Ch. 4] shows that averaging $\zeta_\omega(\mathbf{x})\overline{\zeta_\omega(\mathbf{x}')}$ over D samples of $\boldsymbol{\omega}$ produces a bound on the error of the estimate that diminishes exponentially in D. The lifting map then becomes $\Phi(\mathbf{x}) = \sqrt{1/D}(\zeta_{\omega_1}(\mathbf{x}),\ldots,\zeta_{\omega_D}(\mathbf{x}))$. The inner product $\langle\Phi(\mathbf{x}),\overline{\Phi(\mathbf{x}')}\rangle$ is obviously the desired average.

We can avoid complex numbers by using $z_{\omega,b}(\mathbf{x}) = \sqrt{2}\cos(\boldsymbol{\omega}^\top\mathbf{x}+b)$ with $\boldsymbol{\omega} \sim \mathcal{D}_\mu$ and $b \sim U[0,\pi)$, which offers the same unbiased estimate (see [66]). To see this, consider:

$$\begin{aligned} E_{\omega,b}[z_{\omega,b}(\mathbf{x})z_{\omega,b}(\mathbf{x}')] &= E_{\omega,b}[2\cos(\boldsymbol{\omega}^\top\mathbf{x}+b)\cos(\boldsymbol{\omega}^\top\mathbf{x}'+b)] \\ &= E_{\omega,b}[\cos(\boldsymbol{\omega}^\top(\mathbf{x}+\mathbf{x}')+2b)] + E_\omega[\cos(\boldsymbol{\omega}^\top(\mathbf{x}-\mathbf{x}'))], \end{aligned}$$

from well-known trigonometric identities [1]. Other identities [1] give us

$$\begin{aligned}E_{\omega,b}[\cos(\boldsymbol{\omega}^\top(\mathbf{x}+\mathbf{x}')+2b)] &= E_{\omega}[\cos(\boldsymbol{\omega}^\top(\mathbf{x}+\mathbf{x}'))]E_b[\cos(2b)] \\ &\quad - E_{\omega}[\sin(\boldsymbol{\omega}^\top(\mathbf{x}+\mathbf{x}'))]E_b[\sin(2b)] = 0.\end{aligned}$$

Since the expectation is over an entire period of both functions, both $E_b[\cos(2b)]$ and $E_b[\sin(2b)]$ are zero. So

$$\begin{aligned}E_{\omega,b}[z_{\omega,b}(\mathbf{x})z_{\omega,b}(\mathbf{x}')] &= E_{\omega}[\cos(\boldsymbol{\omega}^\top(\mathbf{x}-\mathbf{x}'))] \\ &= \frac{1}{2}E_{\omega}[e^{i\omega^\top(\mathbf{x}-\mathbf{x}')}] + \frac{1}{2}E_{\omega}[e^{-i\omega^\top(\mathbf{x}-\mathbf{x}')}] \\ &= \frac{1}{2}k(\mathbf{x}-\mathbf{x}') + \frac{1}{2}k(\mathbf{x}'-\mathbf{x}) = k(\mathbf{x}-\mathbf{x}').\end{aligned}$$

The lifting map in this case is $\Phi(\mathbf{x}) = \sqrt{2/D}(z_{\omega_1,b_1}(\mathbf{x}),\ldots,z_{\omega_D,b_D}(\mathbf{x}))$. For more information about this equivalence, see Sutherland and Schneider [71] and Chen and Phillips [19].

In this work, we will refer to these maps (of the real or complex type) as *Fourier embeddings*. In [66], these embeddings are called *random Fourier features*, because they are selected at random from the distribution that is Fourier-dual to the approximated kernel. We will demonstrate that Fourier embeddings of this type need not be selected at random, and can in fact be optimized.

5.2.2.1 Our Approach

Our approach is most similar to that in Băzăvan et al. [17]. Like the authors of [17], we recognize that we can optimize the parameters $\{\boldsymbol{\omega}_i\}$ of a Fourier embedding. Băzăvan et al. decompose $\boldsymbol{\omega}_i$ as follows:

$$\boldsymbol{\omega}_i = \boldsymbol{\sigma}_i \circ h(\mathbf{u}_i),$$

where $\boldsymbol{\sigma}_i$ is the parameter of a shift-invariant kernel, h is an element-wise nonlinear function (essentially an inverse quantile function), and $\mathbf{u}_i$ is a sample drawn from a multivariate uniform distribution (cube). The procedure is to optimize $\boldsymbol{\sigma}_i$ and periodically resample $\mathbf{u}_i$. This has the advantage of being able to represent the kernel with its parameter $\boldsymbol{\sigma}_i$, which adds to clarity, but the kernel must be one of a particular class of shift-invariant kernels that decomposes into this form. A Gaussian kernel, however, does decompose this way.

In contrast, we sample the vectors $\boldsymbol{\omega}_i$ from the distribution $\mathcal{D}_\mu$, and then optimize them directly. The weights $\{\boldsymbol{\omega}_i\}$ become different vectors $\{\boldsymbol{\omega}_i'\} \subset \mathbb{R}^d$ – and are now very unlikely to be drawn i.i.d. from the distribution $\mathcal{D}_\mu$ anymore. As in prior approaches, by learning the embeddings, we learn the kernel, because the Bochner equivalence between distributions and kernels guarantees this. We use backpropagation to learn the weights, avoiding the need to resample at every step, and allowing us to take advantage of recent neural network technology to perform scalable optimization. While other approaches focus on decomposing the representation of the kernels into individual kernel components and learn their parameters, we avoid this and focus only on producing the final weights $\boldsymbol{\omega}'_i$. We lose the clarity and sparsity of individual kernel parameters but gain the flexibility of learning a representation of a shift-invariant kernel free of individual base kernels, and recent technology allows us to do this training quickly.

For brevity, we refer to the $d \times D$ matrices $\mathbf{W}$ (for the $\{\boldsymbol{\omega}_i\}$) and $\mathbf{W}'$ (for the $\{\boldsymbol{\omega}'_i\}$), since there are D samples from $\mathbb{R}^d$.

5.2.3 Generalization Bounds in Fourier Embeddings

We now examine the capacity of this class of kernels by analyzing its VC-dimension. Note that the cosine function complicates this analysis since it has nontrivial gradient almost everywhere.

Fortunately, we can exploit an observation already well-known in kernel learning that a narrow kernel function, for example, a Gaussian kernel with a small variance, is more likely to overfit (and therefore have higher capacity). This is because a narrow kernel function only allows the model to examine a very small range around each point, so a new point is unlikely to be affected by the model at all. Because the kernel is the Fourier transform of a distribution, a narrow kernel function corresponds to a distribution with high variance – using the same example, a Gaussian kernel with variance parameter σ^2 is the Fourier transform of a Gaussian distribution with variance $1/\sigma^2$. So a small variance in the kernel corresponds to a high variance in the distribution, and vice-versa. In fact, we can demonstrate that if the norm of the embedding parameter ω is high, then this translates to higher capacity.

Let $z(x) = e^{2\pi i x}$, $\mathsf{Re}(z)$ and $\mathsf{Im}(z)$ be the real and imaginary components of z, respectively, let $[a..b]$ refer to the set of integers between a and b, inclusive (i.o.w., $\{n \in \mathbb{Z} \mid a \leq n \leq b\}$), and let $\mathbf{1}_P(x)$ be the indicator (or characteristic) function of $P : \mathbb{R} \to \{0, 1\}$.

Definition 3. An $(\boldsymbol{\omega}, \beta, d)$*-range* is the set

$$\{\mathbf{x} \in \mathbb{R}^d \mid \mathsf{Im}(z(\boldsymbol{\omega} \cdot \mathbf{x} + \beta)) \geq 0, \|\mathbf{x}\| < 1\},$$

where $d \geq 1$ is an integer, $\boldsymbol{\omega} \in \mathbb{R}^d$, and $\beta \in [0, 1)$.

Definition 4. Let $\mathcal{G}_d(R)$ be the set of all $(\boldsymbol{\omega}, \beta, d)$-ranges such that $\|\boldsymbol{\omega}\|_2 \leq R$.

Lemma 3. *The decision function* $\mathbf{1}_{\mathsf{Im}(z(wx+\beta))\geq 0}$ *induces a unique binary labeling for the set* $x \in \{1/2^i\}_{i=1}^n$ *for every integer value of* $w \in [1..2^n]$*, and any* $\beta \in (0, 2^{-(n+1)})$*.*

Proof. For any integer $w \in [1..2^n]$ and $i \in [1..n]$, choose the binary label as 0 if $z(w/2^i + \beta)$ lands in the upper half-plane of $\mathbb{C}$, and 1 if the lower half-plane. The label can be read as the most significant fractional digit of the binary representation of $w/2^i$, as long as $\beta \in (0, 2^{-(n+1)})$[2]. The labeling is then unique for integer values of w up to 2^n. □

Clearly, every $(\boldsymbol{\omega}, \beta, d)$-range corresponds to a binary classifier and the range space $(\mathbb{R}^d, \mathcal{G}_d(R))$ is the hypothesis space of interest. We denote the unbounded range space $\cup_R \mathcal{G}_d(R)$ by $\mathcal{G}_d(\infty)$.

Theorem 4. *The VC-dimension of the range space* $(\mathbb{R}^d, \mathcal{G}_d(R))$ *is* $\Theta(\max\{d \log R, d+1\})$*.*

We prove **Theorem 4** in two parts.

Lemma 5. *The VC-dimension of* $(\mathbb{R}^d, \mathcal{G}_d(R))$ *is at least* $d \max\{\lfloor \log_2 R \rfloor, 1\} + 1$*.*

Proof. Let $n = \lfloor \log_2 R \rfloor$, for $R \geq 2$. We now construct a set of dn points. Along each axis of $\mathbb{R}^d$, place n points with corresponding coordinate from the set $\{1/2^i\}_{i=1}^n$. From **Lemma 3**, we know that we can induce a binary labeling on every axis-restricted set, using integers $[1..2^n]$. Given $\boldsymbol{\omega} \in [1..2^n]^d$, each $\omega_j \in [1..2^n]$ will give a unique labeling to the points on axis $j \in [1..d]$, independent of any other axis j. Therefore, we can uniquely label the whole set of dn points, for all possible labelings.

To add one more point to the set, we select a point $\mathbf{c}$, the d-dimensional vector with all coordinates equal to a constant c, and make sure that we can find values β_+ and β_- so that $\langle \mathbf{c}, \boldsymbol{\omega} \rangle + \beta_+ \geq 0$ and $\langle \mathbf{c}, \boldsymbol{\omega} \rangle + \beta_- < 0$, independently of $\boldsymbol{\omega}$. Observe that $\langle \mathbf{c}, \boldsymbol{\omega} \rangle =$

[2]To avoid ambiguity, we require $\beta > 0$, to prevent $z(w/2^i)$ from landing on the real axis when 2^i divides w.

$c\sum_j \omega_j$, and that $d \leq \sum_j \omega_j \leq d2^n$. For $\langle \mathbf{c}, \boldsymbol{\omega} \rangle + \beta_- < 0$ we need that $\beta_+ < -\langle \mathbf{c}, \boldsymbol{\omega} \rangle$ for all $\boldsymbol{\omega}$, since the choice of β must be independent of $\boldsymbol{\omega}$. This means that first, $c < 0$ since $\beta_- > 0$ and $\sum_j \omega_j > 0$. Then $-cd \leq -\langle \mathbf{c}, \boldsymbol{\omega} \rangle \leq -cd2^n$, so we need to pick $\beta_+ < -cd$. Similarly, we require $\beta_+ \geq -cd2^n$, and since $\beta_+ < 2^{-(n+1)}$, we need $-c < 1/d2^{-(2n+1)}$. Set $c = -1/d2^{2n+2}$, $\beta_+ = 2^{-(n+2)}$, and $\beta_- = 2^{-(2n+3)}$. We can now uniquely label $dn + 1$ points for all possible labelings, when $R > 2$.

Regardless of the value of R, there is always a unique labeling of $d + 1$ points induced by the range space, since we can restrict to a ball small enough that $\mathsf{Im}(z(\omega x + \beta)) = \sin(2\pi(\omega x + \beta))$ is monotonic for appropriate values of β. Within the ball, the range space is effectively the range of half-spaces, which has VC-dimension $d + 1$. □

Corollary 6. *The VC-dimension of the range space $(\mathbb{R}^d, \mathcal{G}_d(\infty))$ is unbounded.*

To prove the corresponding upper bound, we use the notion of the *shatter function* of $(\mathbb{R}^d, \mathcal{G}_d(R))$ [38]. For a positive integer n, the shatter function of a range space is the maximum highest number of subsets induced by the range space on any set of n points X_n. That is, any range $\mathcal{R}$ induces a subset of X_n simply by the intersection $\mathcal{R} \cap X_n$, and the shatter function counts all unique subsets of this type.

Lemma 7. *The shatter function of $(\mathbb{R}^d, \mathcal{G}_d(R))$ is $O(R^d n^{d+1})$.*

Proof. We can first observe that $\|\boldsymbol{\omega}\|_2 \leq R$ implies that $\|\boldsymbol{\omega}\|_\infty \leq R$. This implies that $|\omega_j| \leq R$ for every $j \in [1..d]$. Treating each coordinate separately this way, each term in $\langle \boldsymbol{\omega}, \mathbf{x} \rangle + \beta$ contributes a factor in the growth function.

For a fixed $\boldsymbol{\omega}$, the number of subsets of a set of n points selected by $(\boldsymbol{\omega}, \beta, d)$-ranges is $O(n)$, because as β changes, at most one point exits or leaves the upper half-plane (because the points all travel at the same speed around the unit circle).

For fixed β, and fixed $\boldsymbol{\omega}$ save for some coordinate ω_j, on the other hand, how often a point enters or leaves the upper half-plane as ω_j varies in $(0, R]$ depends upon the value of x_j. For higher values of x_j, the mapped point travels more rapidly. In fact, for $x = 1$, z takes R revolutions around the circle, and so enters and exits the upper half-plane $2R$ times. The number of subsets is bounded by

$$\sum_{i=1}^{n} 2R|x_i| = 2R \sum_{i=1}^{n} |x_i| \leq 2Rn.$$

We take the absolute value because a negative x_i simply changes the direction of travel of $z(\omega_j x_i + \beta)$. Everything else remains the same. For $\boldsymbol{\omega}$ and β varying independently, we now have the bound stated in the lemma. □

Lemma 8. *The VC-dimension of* $(\mathbb{R}^d, \mathcal{G}_d(R))$ *is* $O(d \log R)$.

Proof. Follows directly from the relationship between the shatter function and VC dimension [38]. □

With **Lemma 5** and **Lemma 8**, we have proven **Theorem 4**. The VC dimension also gives us a generalization bound, due to Bartlett and Mendelson [8]:

Theorem 9. *Let F be a class of ± 1-valued functions defined on a set $\mathcal{X}$. Let P be a probability distribution on $\mathcal{X} \times \{\pm 1\}$, and suppose that $(X_1, Y_1), \ldots, (X_n, Y_n)$ and (X, Y) are chosen independently according to P. Then for any positive integer n, w.p. $(1-\delta)$ over samples of length n, every $f \in F$ satisfies*

$$P(Y \neq f(X)) \leq \frac{1}{n} \sum_{i=1}^{n} \mathbf{1}_{Y_i \neq f(X_i)} + O\left(\sqrt{\frac{\max\{d \log R, d+1\}}{n}} + \sqrt{\frac{\ln 1/\delta}{n}}\right)$$

5.2.4 Regularization

Theorem 4 and **Theorem 9** immediately suggest a broad regularization strategy: lowering R will lower the sample complexity of the hypothesis class. Intuitively, R places an upper bound on the variance of the distribution dual to the kernel. The signal equivalent to variance is *bandwidth*, which is dual to *frequency* under the Fourier transform. Effectively then by limiting the *variance of the distribution*, we limit the *frequency of the kernel*. We already know this to be a desirable property of kernels — kernels with small effective support produce models that generalize poorly. By forcing our kernel to have broad support, we know that it will generalize better.

At least three regularization techniques then suggest themselves:

- First, we can limit the norm of the Fourier weights with weight decay (a.k.a. L_2 regularization). This is a fairly "smooth" way to control the capacity, because in any iteration, all ω will be scaled by the same amount. This method tends to have a conservative effect on the capacity, since any large change in any *one* ω will scale all the ωs down.

- Alternatively, we can simply cap the norm of each Fourier weight vector to some constant at each round of the training. This is "harsher" than weight decay, because technically, this technique introduces a discontinuity in the distribution. In reality, this is an effective and simple technique.

- We can further control the initial capacity by setting the variance of the initializing distribution. By not setting variance too large, the frequency of the initial kernel will be limited.

5.3 From an Embedding to a Feed-forward Network

We now return to the single Fourier embedding

$$z_{\omega,b} = \sqrt{2}\cos(\boldsymbol{\omega}^\top \mathbf{x} + b)$$

If we fix an input $\mathbf{x}$, then we can view the mapping $z_{\omega,b}$ as a neuron with a cosine activation function and biases of the form $b \in [0, 2\pi)$. We call this type of neuron a *cosine neuron*. Such a neuron, with a cutoff to ensure zero support outside an interval, was introduced in [29]. We impose no such cutoff in this work.

Consider a (hidden) layer of cosine neurons, h_0, each with associated weight vector $\boldsymbol{\omega}_j$. Each of these weights can be viewed as a sample from *some* distribution, and therefore, the entire ensemble is a (dual) representation of some shift-invariant kernel (by Bochner's theorem [14]). We can then write the associated classifier for such a combination. Let us denote the bias vector by $\mathbf{b}_{h_0}$ ($1 \times D$) and the matrix of all the weight vectors $\boldsymbol{\omega}_j$ by $\mathbf{W}_{h_0}$ ($d \times D$). We add a softmax layer for classification, o, with bias $\mathbf{b}_o$ ($1 \times \{\text{\# of classes}\}$) and weights $\mathbf{W}_o$ ($D \times \{\text{\# of classes}\}$). With logarithmic loss to measure the alignment of the classifier with ground truth, we can write:

$$\ell_{\log}(\text{softmax}(\cos(\mathbf{x_i}\mathbf{W}_{h_0} + \mathbf{b}_{h_0})\mathbf{W}_o + \mathbf{b}_o), y_i),$$

where $\ell_{\log}$ is the log loss, and cos is taken elementwise.

What we now have is a standard (shallow) 2-layer network that we can train using backpropagation and stochastic gradient descent.

5.4 Experiments

We have designed our experiments to answer the following questions: (1) Does allowing the learning algorithm to pick an arbitrary kernel improve performance over standard MKL techniques that are only allowed to select from a fixed library of kernels? (2) How does the learning algorithm for CKL adapt to large datasets and higher dimensions?

5.4.1 MKL vs. CKL on Small Datasets

Since CKL is proposed as an alternative to MKL, we compare CKL to two scalable MKL algorithms, namely SPG-GMKL [40] and MWUMKL [57].

5.4.1.1 Datasets

All of the datasets used for the experiments are taken from the `libsvm` repository[3]. See **Table 5.1** for details of the datasets.

The data for Adult and Mushroom datasets consist of binary features (one-hot repre-

[3]`https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary.html`

Table 5.1: Summary of datasets

LibSVM datasets	Features	Examples	
Liver	6	345	
Diabetes	8	768	
Cod-RNA	8	59535	
Breast Cancer	10	683	
German-Numeric	24	1000	
Mushroom	112	8192	
Adult	123	32561	
Gisette	5000	6000	
Million Song Datasets (MSD)	Features	Examples	Notes
Genre 1	182	37,037	*"Classic pop and rock"* vs. *"folk"*
Genre 2	182	59,485	*"Classic pop and rock"* vs. everything else
Year pred.	90	515,345	Prior to year 2000 vs. after year 2000

sentations of categorical features), so no scaling was applied. Features were scaled to the range [-1, 1] for other datasets.

5.4.1.2 Experimental Procedure, MKL

For MKL experiments, we used the *Scikit-Learn* Python package [64] for much of the testing infrastructure. For testing with MKL methods, the training data are split randomly into 75% training and 25% validation data. The random splits were repeated 100 times for all sets except Mushroom, Gisette, and Adult, which received 20 splits for considerations of time. The C parameter was selected through cross validation and for MWUMKL, the ϵ parameter was chosen to be 0.005, to achieve high accuracy while allowing all of the experiments to complete (the number of iterations of the algorithm in [57] is proportional to $1/\epsilon$). We use two kernels: a linear kernel and a Gaussian kernel. For the Gaussian kernel, a wide range of γ are tried and the the best accuracy observed is used in the results.

5.4.1.3 Experimental Procedure, CKL

For CKL experiments, we use the same architectural setup described in Section 5.3 – that is, a hidden layer h_0 that accepts inputs from the dataset and outputs a Fourier embedding, and a softmax layer o that accepts input from the Fourier embedding from h_0 and outputs the prediction of the model. The output of o is evaluated against ground truth using log loss (i.e., multinomial regression), and we attempt to minimize the loss with stochastic gradient descent (SGD) and the backpropagation algorithm.

The same test/train split as in the MKL experiments is applied, and additionally, the training portion is split further into 75% training and 25% validation. We apply early stopping and momentum, and random searches for the following hyperparameters: D (the width of h_0), σ (the initial variance of the weights of h_0), and ℓ (the learning rate). Training was stopped if the validation objective did not decrease within 100 epochs ("early stopping") and was otherwise permitted to run for up to $10,000$ epochs. Momentum was applied from the first epoch with a value of 0.5 that was increased to 0.99 over the course of 10 epochs.

The parameters of the model, that is, the weights of the connections of the network, were initialized randomly. The weights of h_0 were sampled from a normal distribution with variance σ, and the weights of o were selected uniformly from the interval $[-0.1, 0.1]$.

See **Table 5.2**.

Values for D were drawn from $\{2^i \mid i \in [0..9]\}$, except for Gisette, where the selection is from $\{2^i \mid i \in [0..14]\}$. Values for σ were selected from $\{2^i \mid i \in [-6..0]\}$. Finally, ℓ was sampled from $LU[10^{-5}, 0.2)$[4]. See **Table 5.3**. 100 models with random hyperparameters were trained, and then the one with the highest performance was chosen and validated with 100 random splits (as described in the previous paragraph).

Note that while we *did* test both weight decay and norm capping (see Section 5.2.4), the results were inconclusive so they are not reported here.

5.4.1.4 Results

The results are shown in **Table 5.4**. CKL is not different in any significant capacity from either SPG-GMKL or MWUMKL on very small datasets. Letting the learning algorithm pick an arbitrary kernel improves performance over standard MKL techniques that only choose a mixture of kernels. Additionally, we see that CKL adapts to large datasets and higher dimensions better than MKL.

5.4.2 MKL vs. CKL on Million Song Datasets

In this section, we compare MKL methods with CKL on the Million Song Dataset [11]. The Million Song Dataset consists of audio features and metadata of one million contemporary popular music tracks. For the experiments, we utilized three different subsets of the Million Song Dataset, all binary. The features are the average and covariance of the pitch and timbre vectors for each track:

Genre 1: The two most common genres in Million Song Dataset - "*classic pop and rock*" and

[4]A random variable X is drawn from $LU[a, b]$ if $X = e^Y$, where $Y \sim U[\ln(a), \ln(b))$.

Table 5.2: Parameters used in CKL experiments. Note that σ is a hyperparameter of the model.

Parameters	Dimensions	Initial Distribution
$\mathbf{W}_{h_0}$	$d \times D$	$\mathcal{N}(0, \sigma^2)$
$\mathbf{b}_{h_0}$	$1 \times D$	$U[0, \pi)$
$\mathbf{W}_o$	$D \times 2$	$U[-0.1, 0.1]$
$\mathbf{b}_o$	1×2	constant 0

Table 5.3: Hyperparameters used in CKL experiments.

Hyperparameter	Name	Values
Width of h_0	D	2^i, with $i \sim [0..9]$ (except Gisette, where $i \sim [0..14]$)
Variance of elements in $\mathbf{W}_{h_0}$	σ^2	2^i, with $i \sim [-6..0]$
Learning rate	ℓ	$LU[10^{-5}, 0.2)$

Table 5.4: Mean accuracies (standard deviations) for various datasets on MKL and CKL. If a mean, minus the standard deviation, is greater than all other means plus standard deviations in the row, then the mean is bold. Note that for all MSD tests, the difference is more than three standard deviations.

Small Datasets	SPG-GMKL	MWUMKL	CKL
Liver	67.78% (4.78%)	59.34% (6.04%)	66.45% (6.19%)
Diabetes	77.06% (2.66%)	75.59% (2.92%)	76.08% (2.95%)
Cod-RNA	**87.31%** (0.13%)	72.42% (7.30%)	85.7% (1.14%)
Breast Cancer	97.14% (1.20%)	91.89% (2.22%)	96.87% (1.22%)
German-Numeric	73.05% (3.25%)	74.40% (3.01%)	76.14% (2.57%)
Mushroom	99.80% (0.08%)	99.93% (0.04%)	**100%** (0.0042%)
Adult Income	83.94% (0.28%)	76.90% (0.82%)	**84.80%** (0.35%)
Gisette	95.15% (0.53%)	93.50% (0.72%)	**96.90%** (0.52%)
Million Song Dataset	SPG-GMKL	MWUMKL	CKL
Genre 1	77.62% (0.36%)	68.14% (1.06%)	**81.68%** (0.39%)
Genre 2	69.12% (0.33%)	53.02% (0.55%)	**74.16%** (0.36%)
Year Pred.	75.38% (0.1%)	57.72% (1.64%)	**77.57%** (0.11%)

"*folk*." The tracks which have both genres as tags are removed to avoid confusion.

Genre 2: The ten most common genres in the Million Song Dataset. Since the "*classic pop and rock*" genre has significantly more tracks than any other genre, "*classic pop and rock*" is considered as one class and everything else together as another class.

Year Prediction: Taken from the UCI Machine Learning Repository. All tracks prior to the year 2000 are considered as one class and all tracks after and including the year 2000 are considered as the other class. The dimensions of the dataset are summarized in **Table 5.1**.

5.4.2.1 Results

The results are shown in **Table 5.4**. CKL is clearly superior to the scalable MKL methods that we tested against, adding to the evidence that higher-dimensional and larger datasets can benefit from our technique.

5.4.3 MKL vs. CKL on Images

We compare MKL and CKL on CIFAR10. CIFAR10 [47] is a labeled image dataset containing 60,000 1,024-dimensional (32×32) images and 10 classes used extensively for testing image classification algorithms. While image classification is an important benchmark for neural networks, we wish to point out that our objective is *not* to classify the CIFAR10 dataset better than all other previous techniques. Instead, we wish to provide comparisons between the methods described in this chapter on a large and very challenging task using a simple convolutional neural architecture.

5.4.3.1 Preprocessing

We first centered the CIFAR10 training set by mean, and then used Pylearn2 [35] to apply two transformations: global contrast normalization [22] and ZCA whitening [9][5]. We applied the same transformations computed for the training set to the testing set.

5.4.3.2 Feature Extraction

For MKL, we used a convolutional neural network (CNN) [51] to learn a representation from the data. In total, we trained 100 models and we extracted the features from the model with the best performance. All of the models had the form $\text{conv}_{\text{ReLU}} \rightarrow \text{pool}_{max} \rightarrow \text{fc}_{\text{ReLU}} \rightarrow \text{softmax}$ where $\text{conv}_{\text{ReLU}}$ is a convolutional layer using ReLU nonlinearities, pool_{max} is a max-pool layer, fc_{ReLU} was a fully-connected layer using ReLU nonlinearities, and softmax was a *softmax* layer.

We trained the models with (1) momentum, initialized to 0.5 and increased to 0.99 over the first 100 epochs, and (2) early stopping: we set aside the last $10,000$ samples of the

[5]PCA whitening attempts to decorrelate features and normalize singular values ("whitening") of the original data by rotating the data by singular vectors, and then normalizing singular values. ZCA whitening, in contrast, attempts to do the same, but make the resulting data as close to the original as possible, in a least-squares sense. The ZCA transformation is simply to multiply by the inverse square root of the covariance matrix of the data.

training set as a validation set for early stopping, and trained the models for at most $5,000$ epochs. We initialized the weights of all layers by selecting values uniformly at random from the range $[-0.01, 0.01]$. The parameters of the best performing model were as follows: (1) the convolutional layer (with ReLU activations): a 5×5 kernel with 1×1 stride, 32 channels, a max kernel norm of 1.8, and cross channel normalization with $\alpha = 3.2 \times 10^{-4}$ and $\beta = 0.75$, (2) the max pooling layer: a 3×3 kernel with 2×2 stride, (3) the fully connected layer: $1,000$ rectified linear units, and (4) the softmax layer: one output for each CIFAR10 class. Each sample of CIFAR10 was passed through the CNN and the activations of the fully connected layer were recorded as the new representation.

5.4.3.3 CIFAR10 with MKL

For MKL experiments, the testing infrastructure and the experimental procedures are similar to the experimental procedure of Section 5.4.1 except for the following details: (1) One-vs-one multiclass strategy is used for the classification task, (2) Random 75% of the training data is used for training and tested on the standard test data. The runs were repeated 20 times, and (3) We used two Gaussian kernels, one with $\gamma = 1$ and the other with a range of γ from 2^{-7} to 2^{7}. The best accuracy observed is shown in **Table 5.5**.

5.4.3.4 CIFAR10 with CKL

For comparison with MKL, we trained a network of the form $\text{conv}_{\text{ReLU}} \to \text{pool}_{max} \to \text{fc}_{\text{ReLU}} \to \text{fc}_{\cos} \to \text{softmax}$. A CKL model of this form uses the same structure as the CNN used for the MKL/CKL experiments (defined in Section 5.4.3.2), up to and including the fully connected layer of rectified linear units. Instead of a softmax layer, the units of the fully connected layer were connected to a CKL model with $1,000$ hidden units (untuned).

The primary difference between this model and MKL trained on features extracted from a CNN (see Section 5.4.3.3) is that this model is trained all at once, while in the MKL experiments, the CNN used for feature learning and the MKL model were trained sepa-

Table 5.5: Accuracy for CIFAR10 on MKL and CKL with CNN.

SPG-GMKL	MWUMKL	CKL+CNN
44.43% (0.57%)	48.2% (0.41%)	**67.77%** (0.61%)

rately. This end-to-end learning allows the features of each layer to adapt to the features that appear later in the network. It is also important to note that the MKL experiments were trained on a one-vs.-one basis, while the CKL model uses multinomial (softmax) regression with log loss.

5.4.3.5 Experimental Procedure

The models in these experiments were trained using stochastic gradient descent for a maximum of $1,000$ epochs with early stopping and momentum. The initial momentum rate was 0.5 and was adjusted from the first epoch to 0.99 over the first 500 epochs of the training.

5.4.3.6 Results

The CKL model outstrips the MKL methods by a wide margin. We conjecture that this is due to two effects: (1) the end-to-end training allows for better adaptation in the training process and (2) the search space of kernels is much larger. The first effect demonstrates that CKL is more adaptable than MKL in these settings. It is also important to note that training is a crucial component for CKL models when operating on large datasets. For CIFAR10, evaluating any random model upon initialization yielded an accuracy of only 10.1% with standard deviation of 0.235%. In contrast, evaluating random models on smaller datasets frequently yields accuracies that are better than chance.

5.4.3.7 CIFAR10 with Two Layer ConvNets

One might ask whether stacking two cosine layers has any beneficial effect, since stacking two cosine layers is similar to composing two lifting maps, which if defined, yields a kernel. Zhuang et al. [87] construct an algorithm specifically for the composition of two kernels – essentially layering the kernels. Lu et al. [54] discuss extensions to [66] that cover products, sums, and compositions of kernels. Since these are based on the sampling methodology of [66], there is a direct analogy to composing two cosine layers (fixed, in this case). We did not observe significant improvement in accuracy when we employed combinations of two cosine layers. One possible explanation is that since the composition of a kernel is itself a kernel, it can be argued that optimizing a network that contains two consecutive cosine layers accomplishes no more than doing so with one cosine layer.

CHAPTER 6

CONCLUSION

We have explored just a few aspects of kernel learning – optimization, localization, and distributional. We have demonstrated that kernel learning continues to be a rich area for research.

6.1 Summary of Contributions and Future Directions

6.1.1 Multiplicative Weight Updates-MKL

We presented a simple, fast, and easy to implement algorithm for multiple kernel learning (MKL). 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 with Nyström kernel approximations (LibLinear+) as well as uniformly weighted combination of kernels (Uniform).

Our current results are for a single machine. One area of current research has been to add parallellization techniques to improve the scalability of MWUMKL over datasets that are large and use a large number of kernels. The MWUMKL algorithm lends itself easily to the *bulk synchronous parallel* (BSP) framework [72], as most of the work is done in the loop that updates $\mathbf{G}\alpha$ (see the last line of the loop in **Algorithm 6**). 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.

6.1.2 Localized Kernel Learning

We analyzed several localized kernel learning (LKL) algorithms, and developed a unification of the ideas that they contain. We then developed a new algorithm based upon those ideas that is efficient and accurate. We also analyzed the geometry of unified LKL reproducing kernel Hilbert spaces (RKHSs), and empirically supported the proposition that LKL algorithms produce fewer support points.

Our current research in this area is to develop an extension to LD-MKL that performs regression on data, not simply classification. While the extension is simple in theory, a few caveats must be satisfied, such as choosing a method to measure “success.”

6.1.3 Continuous Kernel Learning

We depart from the support vector machine (SVM)-based kernel learning techniques of the previous two sections and develop a framework of learning a kernel embedded in a neural net with backpropagation. Importantly, we distinguish our results from other Bochner's theorem [14] work by proving a sample complexity bound for cosine learners, and use this result to argue for several regularization techniques (which are vitally important when using neural nets). We compare cosine nets to previous MKL results, and find that our new technique is significantly superior, especially with respect to image data.

Future work is wide open in this area — as many techniques and approaches can be tested against continuous kernel learning (CKL) as there are new techniques in deep learning. Examples include applying cosine nets as part of recurrent neural networks or generative adversarial networks. In addition, we have only begun to test various kinds of data, and it would be interesting to see if data from realms that use Fourier mathematics, such as signal processing, could benefit better from our technique.

DISSEMINATION OF THIS WORK

- **A Geometric Algorithm for Scalable Multiple Kernel Learning**
  John Moeller, Parasaran Raman, Avishek Saha, and Suresh Venkatasubramanian
  17th International Conference on Artificial Intelligence and Statistics (AISTATS)
  Reykjavík, Iceland; 2014

- **A Unified View of Localized Kernel Learning**
  2016 SIAM International Conference on Data Mining (SDM)
  John Moeller, Sarathkrishna Swaminathan, and Suresh Venkatasubramanian
  Miami, Florida, USA; May 2016

- **Continuous Kernel Learning**
  2016 European Conference on Machine Learning and Principles and Practice of Knowledge Discovery (ECMLPKDD)
  John Moeller, Vivek Srikumar, Sarathkrishna Swaminathan, Suresh Venkatasubramanian, and Dustin Webb
  Riva di Garda, Italy; September 2016

REFERENCES

[1] Milton Abramowitz and Irene A Stegun. *Handbook of mathematical functions: with formulas, graphs, and mathematical tables*, volume 55. Courier Corporation, 1964.

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

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

[4] N. Aronszajn. Theory of reproducing kernels. *Transactions of the American Mathematical Society*, 68(3):337–404, 1950.

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

[6] Özlem Aslan, Xinhua Zhang, and Dale Schuurmans. Convex deep learning via normalized kernels. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *NIPS*, pages 3275–3283. Curran Associates, Inc., 2014.

[7] Francis R. Bach, Gert R. G. Lanckriet, and Michael I. Jordan. Multiple kernel learning, conic duality, and the SMO algorithm. In *ICML*, pages 6–13, New York, NY, USA, 2004. ACM.

[8] Peter L. Bartlett and Shahar Mendelson. Rademacher and Gaussian complexities: Risk bounds and structural results. *JMLR*, 3:463–482, March 2003.

[9] Anthony J. Bell and Terrence J. Sejnowski. Edges are the 'independent components' of natural scenes. In M. C. Mozer, M. I. Jordan, and T. Petsche, editors, *NIPS*, pages 831–837. MIT Press, 1997.

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

[11] Thierry Bertin-Mahieux, Daniel P. W. Ellis, Brian Whitman, and Paul Lamere. The million song dataset. In *ISMIR*, 2011.

[12] Liefeng Bo, Xiaofeng Ren, and Dieter Fox. Kernel descriptors for visual recognition. In J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta, editors, *NIPS*, pages 244–252. Curran Associates, Inc., 2010.

[13] Liefeng Bo, K. Lai, Xiaofeng Ren, and D. Fox. Object recognition with hierarchical kernel descriptors. In *CVPR*, pages 1729–1736, June 2011.

[14] Salomon Bochner. *Lectures on Fourier integrals*. Number 42 in Annals of Mathematics Studies. Princeton University Press, 1959.

[15] Bernhard E. Boser, Isabelle M. Guyon, and Vladimir N. Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, COLT '92, pages 144–152, New York, NY, USA, 1992. ACM.

[16] S.P. Boyd and L. Vandenberghe. *Convex optimization*. Cambridge Univ Press, 2004.

[17] E. G. Băzăvan, F. Li, and C. Sminchisescu. Fourier kernel learning. In *ECCV*, 2012.

[18] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. *ACM TIST*, 2:27:1–27:27, May 2011. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[19] Di Chen and Jeff M Phillips. Relative error embeddings for the Gaussian kernel distance. *arXiv preprint arXiv:1602.05350*, 2016.

[20] Radha Chitta, Rong Jin, Timothy C Havens, and Anil K Jain. Approximate kernel k-means: Solution to large scale kernel clustering. In *KDD*, pages 895–903. ACM, 2011.

[21] Youngmin Cho and Lawrence K. Saul. Kernel methods for deep learning. In Y. Bengio, D. Schuurmans, J. D. Lafferty, C. K. I. Williams, and A. Culotta, editors, *NIPS*, pages 342–350. Curran Associates, Inc., 2009.

[22] Adam Coates, Andrew Y. Ng, and Honglak Lee. An analysis of single-layer networks in unsupervised feature learning. In *AIStats*, pages 215–223, 2011.

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

[24] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20 (3):273–297, 1995.

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

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

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

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

[29] A.R. Gallant and H. White. There exists a neural network that does not make avoidable mistakes. In *ICNN*, pages 657–664 vol.1, July 1988.

[30] Bernd Gärtner and Martin Jaggi. Coresets for polytope distance. In *Proceedings of the Twenty-fifth Annual Symposium on Computational Geometry*, SCG '09, pages 33–42, New York, NY, USA, 2009. ACM.

[31] Elmer G. Gilbert. An iterative procedure for computing the minimum of a quadratic form on a convex set. *SIAM Journal on Control*, 4(1):61–80, 1966.

[32] Amir Globerson and Roi Livni. Learning infinite-layer networks: Beyond the kernel trick. *arXiv:1606.05316 [cs]*, June 2016. URL `http://arxiv.org/abs/1606.05316`. arXiv: 1606.05316.

[33] Mehmet Gönen and Ethem Alpaydin. Localized multiple kernel learning. In *ICML*, pages 352–359, 2008.

[34] Mehmet Gönen and Ethem Alpaydin. Localized algorithms for multiple kernel learning. *Pattern Recognition*, 46(3):795–807, 2013.

[35] Ian J. Goodfellow, David Warde-Farley, Pascal Lamblin, Vincent Dumoulin, Mehdi Mirza, Razvan Pascanu, James Bergstra, Frédéric Bastien, and Yoshua Bengio. Pylearn2: a machine learning research library. *arXiv:1308.4214 [cs, stat]*, August 2013. URL `http://arxiv.org/abs/1308.4214`. arXiv: 1308.4214.

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

[37] Yina Han and Guizhong Liu. Probability-confidence-kernel-based localized multiple kernel learning with norm. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, 42(3):827–837, 2012.

[38] Sariel Har-Peled. *Geometric Approximation Algorithms*. American Mathematical Society, Boston, MA, USA, 2011.

[39] Tamir Hazan and Tommi Jaakkola. Steps toward deep kernel methods from infinite neural networks. *arXiv:1508.05133 [cs]*, August 2015. URL `http://arxiv.org/abs/1508.05133`. arXiv: 1508.05133.

[40] Ashesh Jain, S. V. N. Vishwanathan, and Manik Varma. SPG-GMKL: generalized multiple kernel learning with a million kernels. In *KDD*, pages 750–758, 2012.

[41] M. Jiu and H. Sahbi. Deep kernel map networks for image annotation. In *ICASSP*, pages 1571–1575, March 2016.

[42] M. Jiu and H. Sahbi. Laplacian deep kernel learning for image annotation. In *ICASSP*, pages 1551–1555, March 2016.

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

[44] Raghvendra Kannao and Prithwijit Guha. *TV Commercial Detection Using Success Based Locally Weighted Kernel Combination*, pages 793–805. Springer International Publishing, Cham, 2016.

[45] Marius Kloft, Ulf Brefeld, Soeren Sonnenburg, Pavel Laskov, Klaus-Robert Müller, and Alexander Zien. Efficient and accurate lp-norm multiple kernel learning. In *NIPS*, Vancouver, Canada, 2009.

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

[47] Alex Krizhevsky. *Learning multiple layers of features from tiny images*. Citeseer, 2009.

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

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

[50] Quoc Le, Tamas Sarlos, and Alexander Smola. Fastfood - computing hilbert space expansions in loglinear time. In *ICML*, pages 244–252, 2013.

[51] Y. LeCun. Generalization and network design strategies. In R. Pfeifer, Z. Schreter, F. Fogelman, and L. Steels, editors, *Connectionism in Perspective*, Zurich, Switzerland, 1989. Elsevier. an extended version was published as a technical report of the University of Toronto.

[52] Yunwen Lei, Alexander Binder, Ürün Dogan, and Marius Kloft. Localized multiple kernel learning - a convex approach. *CoRR*, abs/1506.04364, 2015. URL `http://arxiv.org/abs/1506.04364`.

[53] Xinwang Liu, Lei Wang, Jian Zhang, and Jianping Yin. Sample-adaptive multiple kernel learning. In *AAAI*, 2014.

[54] Zhiyun Lu, Avner May, Kuan Liu, Alireza Bagheri Garakani, Dong Guo, Aurélien Bellet, Linxi Fan, Michael Collins, Brian Kingsbury, Michael Picheny, and Fei Sha. How to scale up kernel methods to be as good as deep neural nets. *arXiv:1411.4000 [cs, stat]*, November 2014. arXiv: 1411.4000.

[55] Julien Mairal, Piotr Koniusz, Zaid Harchaoui, and Cordelia Schmid. Convolutional Kernel Networks. In *NIPS*, pages 2627–2635, 2014.

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

[57] John Moeller, Parasaran Raman, Suresh Venkatasubramanian, and Avishek Saha. A geometric algorithm for scalable multiple kernel learning. In *AIStats*, pages 633–642, 2014.

[58] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms:*. Cambridge University Press, Cambridge, 008 1995. doi: 10.1017/CBO9780511814075.

[59] Radford M. Neal. Priors for infinite networks. In *Bayesian Learning for Neural Networks*, number 118 in Lecture Notes in Statistics, pages 29–53. Springer New York, 1996.

[60] Junier Oliva, Avinava Dubey, Barnabas Poczos, Jeff Schneider, and Eric P. Xing. Bayesian nonparametric kernel-learning. *arXiv:1506.08776 [stat]*, June 2015. URL `http://arxiv.org/abs/1506.08776`. arXiv: 1506.08776.

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

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

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

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

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

[66] Ali Rahimi and Benjamin Recht. Random features for large-scale kernel machines. In *NIPS*, pages 1177–1184, 2007.

[67] Alain Rakotomamonjy, Francis Bach, Stéphane Canu, and Yves Grandvalet. More efficiency in multiple kernel learning. In *ICML*, Corvalis, USA, 2007.

[68] Bernhard Schölkopf, Ralf Herbrich, and Alex J. Smola. *A Generalized Representer Theorem*, pages 416–426. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

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

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

[71] Dougal J Sutherland and Jeff Schneider. On the error of random Fourier features. *UAI*, pages 862–871, July 2015.

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

[73] Vladimir Vapnik and Alexey Chervonenkis. A note on one class of perceptrons. *Automation and remote control*, 25(1), 1964.

[74] Manik Varma and Bodla Rakesh Babu. More generality in efficient multiple kernel learning. In *ICML*, Montreal, Canada, 2009.

[75] S. V. N. Vishwanathan, Zhaonan Sun, Nawanol Ampornpunt, and Manik Varma. Multiple kernel learning and the SMO algorithm. In *NIPS*, Vancouver, Canada, 2010.

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

[77] Andrew Wilson and Ryan Adams. Gaussian process kernels for pattern discovery and extrapolation. In *ICML*, pages 1067–1075, 2013.

[78] Andrew Gordon Wilson, Zhiting Hu, Ruslan Salakhutdinov, and Eric P. Xing. Deep kernel learning. *arXiv:1511.02222 [cs, stat]*, November 2015. URL `http://arxiv.org/abs/1511.02222`. arXiv: 1511.02222.

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

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

[81] Jingjing Yang, Yuanning Li, Yonghong Tian, Lingyu Duan, and Wen Gao. Group-sensitive multiple kernel learning for object categorization. In *ICCV*, pages 436–443. IEEE, 2009.

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

[83] Zichao Yang, Marcin Moczulski, Misha Denil, Nando de Freitas, Alex Smola, Le Song, and Ziyu Wang. Deep fried convnets. *arXiv:1412.7149 [cs, stat]*, December 2014. URL `http://arxiv.org/abs/1412.7149`. arXiv: 1412.7149.

[84] Zichao Yang, Andrew Wilson, Alex Smola, and Le Song. À la carte – learning fast kernels. In *AIStats*, pages 1098–1106, 2015.

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

[86] Felix X. Yu, Sanjiv Kumar, Henry Rowley, and Shih-Fu Chang. Compact nonlinear maps and circulant extensions. *arXiv:1503.03893 [cs, stat]*, March 2015. URL `http://arxiv.org/abs/1503.03893`. arXiv: 1503.03893.

[87] Jinfeng Zhuang, Ivor W. Tsang, and Steven Hoi. Two-layer multiple kernel learning. In *AIStats*, pages 909–917, 2011.

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