High-performance Kernel Machines with Implicit Distributed Optimization and Randomization

Vikas Sindhwani* Haim Avron†

Abstract

Complex machine learning tasks arising in several domains increasingly require "big models" to be trained on "big data". Such models tend to grow with the complexity and size of the training data, and do not make strong parametric assumptions upfront on the nature of the underlying statistical dependencies. Kernel methods constitute a very popular, versatile and principled statistical methodology for solving a wide range of non-parametric modelling problems. However, their storage requirements and high computational complexity poses a significant barrier to their widespread adoption in big data applications. We propose an algorithmic framework for massive-scale training of kernel-based machine learning models. Our framework combines two key technical ingredients: (i) distributed general purpose convex optimization for a class of problems involving very large but implicit datasets, and (ii) the use of randomization to significantly accelerate the training process as well as prediction speed for kernel-based models. Our approach is based on a block-splitting variant of the Alternating Directions Method of Multipliers (ADMM) which is carefully reconfigured to handle very large random feature matrices only implicitly, while exploiting hybrid parallelism in compute environments composed of loosely or tightly coupled clusters of multicore machines. Our implementation supports a variety of machine learning tasks by enabling several loss functions, regularization schemes, kernels, and layers of randomized approximations for both dense and sparse datasets, in a highly extensible framework. We study the scalability of our framework on both commodity clusters as well as on BlueGene/Q, and provide a comparison against existing sequential and parallel libraries for such problems.

Key Words: Kernel Methods, High-performance Computing, ADMM, Sketching.

1. Introduction

A large class of supervised machine learning models are trained by solving optimization problems of the form,

$$f^\star = \underset{f \in \mathcal{H}}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} V\left(\mathbf{y}_i, f(\mathbf{x}_i)\right) + \lambda r(f), \tag{1.1}$$

where,

- $\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^n$ is a training set with n labeled examples, with inputs $\mathbf{x} \in \mathcal{X} \subset \mathbb{R}^d$ and associated target outputs $\mathbf{y} \in \mathcal{Y} \subset \mathbb{R}^m$;
- $\mathcal{H}$ is a hypothesis space of functions mapping the input domain in $\mathbb{R}^d$ to the output domain in $\mathbb{R}^m$, over which the training process estimates a functional dependency $f^\star$ by optimizing an objective function;
- the objective function comprises of a convex loss function $V(\cdot, \cdot)$ which measures the discrepancy between "ground truth" and model predictions, and a convex regularizer $r(\cdot)$ that penalizes the complexity of f in order to prevent the phenomenon of overfitting. The regularization parameter λ balances the classic tradeoff between data fitting and complexity control in machine learning, which enables generalization to unseen test data.

*Machine Learning Group, IBM Research, NY 10598. Email: vsindhw@us.ibm.com

†HPC for Analytics Group, IBM Research NY 10598. Email: haimav@us.ibm.com

Modern "big data", arising in several domains of significant commercial and academic interest, is characterized by large training sets (big n) often also in conjunction with high input and output data dimensionality (big d, big m). It is now well appreciated among machine learning researchers and practitioners, that imposing strong structural constraints on the model upfront, e.g. by requiring $\mathcal{H}$ to comprise of linear functions, or otherwise severely restricted in terms of sparsity or smoothness, often limits, both theoretically and empirically, the potential of big data in terms of delivering higher accuracy models. In such cases, data tends to exhausts the statistical capacity of the model causing generalization performance to quickly saturate. As a consequence, machine learning practitioners are increasingly turning to highly nonlinear models with millions of parameters, or even infinite-dimensional models that need to be estimated on very large datasets [1, 2, 3, 4, 5], often with carefully designed domain-dependent loss functions and regularizers. This trend is giving rise to new challenges and opportunities at the intersection of statistics, numerical optimization and high performance computing.

Kernel methods [6] constitute a mathematically elegant framework for general-purpose infinite-dimensional non-parametric statistical inference . By providing a principled framework to extend classical linear statistical techniques for non-parametric modeling, their applications span the entire spectrum of machine learning: nonlinear classification, regression, clustering, time-series analysis, sequence modeling [7], dynamical systems [8], hypothesis testing [9], causal modeling [10] and others. As such, with the growth in data across a multitude of applications, scaling up kernel methods [11] has acquired renewed and somewhat urgent significance.

The central object in kernel methods is a kernel function $k(\mathbf{x}, \mathbf{x}')$ defined on the input domain $\mathcal{X}$. This kernel function defines a suitable hypothesis space of functions $\mathcal{H}$ with which (1.1) can be instantiated, and turned into a finite dimensional optimization problem. However, training procedures derived directly in this manner scale poorly, having training time that is cubic in n and storage that is quadratic in n, with limited opportunities for parallelization.

Recently, randomization has emerged as a key algorithmic device with which to dramatically accelerate the training of kernel methods. Randomized methods replace the nonlinear modeling problem (1.1), with a linear modeling problem of the form,

$$\underset{\mathbf{W}\in\mathbb{R}^{s\times k}}{\operatorname{argmin}} \frac{1}{n}\sum_{i=1}^{n} V\left(\mathbf{y}_i, \mathbf{W}^T\mathbf{z}(\mathbf{x}_i)\right) + \lambda r(\mathbf{W}) \tag{1.2}$$

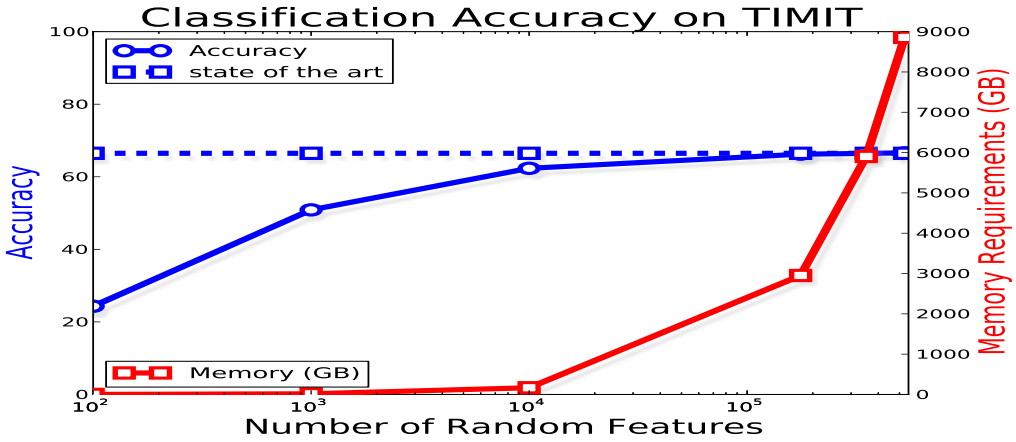
where $\mathbf{W}$ are coefficients of a linear model to be estimated from the training data. The function $\mathbf{z}(\cdot)$ is a nonlinear random transformation of the data that replaces the original input point $\mathbf{x}$ with a point in a higher-dimensional nonlinear random feature space, $\mathbf{z}(\mathbf{x}_i) \in \mathbb{R}^s$. This transformation has the property that with high probability $\mathbf{z}(\mathbf{x})^T\mathbf{z}(\mathbf{x}')$ is close to the kernel function $k(\mathbf{x}, \mathbf{x}')$ for all $\mathbf{x}, \mathbf{x}'$ in the data domain.

Figure 1.1 motivates this paper. It is observed that in order to build state-of-the-art machine learning models in problems of interest (in this case, speech recognition), a very large number of random features are needed. The transformed data matrix – $\mathbf{Z}$ – whose rows are $z(\mathbf{x}_i)$, becomes terascale even though the original data matrix $\mathbf{X}$ is relative small. We refer to problems of the form in (1.2) as *implicit* optimization problems, involving "big data" which is implicitly actually a function of smaller data.

We are now in a position to state our contributions.

- We develop a highly scalable algorithmic framework and software implementation for kernel methods, for distributed-memory computing environments. Our framework combines randomization with a distributed optimization method based on the Alternating Directions Method of Multipliers (ADMM) [12]. This framework orchestrates local

Figure 1.1: Performance of Randomized Kernel Methods. The largest model below is trained on BlueGene/Q, on an 8.7 terabyte dataset using implicit distributed optimization methods implemented in this paper for high-performance computing environments.



models estimated on a subset of examples and random features, towards the solution of (1.2). Our approach builds on the block-splitting ADMM framework proposed by [13], but we reorganize its update rules to extract greater efficiency.

- Our framework is designed to be highly generalizable. In particular, a user needs to only supply certain proximal operators associated with a custom loss function and the regularizer. Our ADMM wrapper can then immediately instantiate a solver for (1.2) for a variety of choices of kernels and learning tasks.
- We benchmark our implementation in both supercomputing environments as well as commodity clusters. Results indicate that our approach is highly scalable in both settings, and is capable of returning state of the art performance in machine learning tasks. Comparisons against sequential and parallel libraries for Support vector Machines shows highly favorable accuracy-time tradeoffs for our approach.

2. Distributed Learning with ADMM Block Splitting and Hybrid Parallelism

We assume the following setup:

- A distributed computing environment comprising of a cluster of N compute nodes, with T cores per node. We assume that each node has M GigaBytes of RAM.
- The training data $\mathbf{X}, \mathbf{Y}$ are distributed *by rows* across the nodes. This is a natural assumption in machine learning since rows have the semantics of data instances which are typically collected or generated in parallel across the cluster to begin with.
- $\mathbf{X}, \mathbf{Y}$ fit in the aggregate distributed memory of the cluster, but are large enough that they cannot fit on a single node, and cannot be replicated in memory on multiple nodes.
- The matrix $\mathbf{Z}$ does not fit in aggregate distributed memory because n and s are both simultaneously big. This assumption is motivated by empirical observations shown in Figure 1.1.
- The matrix $\mathbf{Z}$ cannot be stored on disk either, because of space restrictions or because the IO cost of reading $\mathbf{Z}$ by blocks in every iteration is more expensive than the cost of recomputing blocks of $\mathbf{Z}$ on the fly from scratch as needed, respecting per-node memory constraints.

We build on a block-splitting variant of the Alternating Direction Method of Multipliers (ADMM) [13] which is well suited to these considerations. This approach assumes the data matrix is partitioned by both rows and columns. Independent models are estimated on each block in parallel and orchestrated by ADMM towards the solution of the optimization problem. A common pattern in these algorithms are the following operators.

Definition 1 (Proximity Operator) *The Proximity (or Prox) operator associated with a function $f : \mathbb{R}^d \mapsto \mathbb{R}$ is a map $\texttt{prox}_{\lambda f} : \mathbb{R}^d \mapsto \mathbb{R}^d$ given by*

$$\texttt{prox}_f[\mathbf{x}] = \underset{\mathbf{y}\in\mathbb{R}^d}{\operatorname{argmin}} \frac{1}{2}\|\mathbf{x}-\mathbf{y}\|_2^2 + f(\mathbf{y}) \tag{2.1}$$

Definition 2 (Projection Operator) *The Projection operator associated with a convex constraint set $\mathcal{C}$ is the map $\texttt{proj}_{\mathcal{C}} : \mathbb{R}^d \mapsto \mathbb{R}^d$ given by*

$$\texttt{proj}_{\mathcal{C}}[\mathbf{x}] = \underset{\mathbf{y}\in\mathcal{C}}{\operatorname{argmin}} \frac{1}{2}\|\mathbf{x}-\mathbf{y}\|_2^2 \tag{2.2}$$

Hence, $\texttt{proj}_{\mathcal{C}} = \texttt{prox}_{\mathbb{I}_{\mathcal{C}}}$ where $\mathbb{I}_{\mathcal{C}}$ denotes indicator function of the set $\mathcal{C}$ i.e. assuming the value 0 on $\mathcal{C}$ and ∞ otherwise.

For a variety of common loss functions and regularizers, the proximal operator admits fast closed-form solutions. In addition to the proximal and projection operators, the block-splitting approach also needs the following definition.

Definition 3 (Graph Projection Operator over Matrices) *The Graph projection operator associated with an $n \times d$ matrix $\mathbf{A}$ is the map $\texttt{proj}_{\mathbf{A}} : (\mathbb{R}^{n\times k}, \mathbb{R}^{d\times k}) \mapsto (\mathbb{R}^{n\times k}, \mathbb{R}^{d\times k})$ given by,*

$$\texttt{proj}_{\mathbf{A}}[(\mathbf{Y},\mathbf{X})] = \underset{\mathbf{V},\mathbf{U}}{\operatorname{argmin}} \frac{1}{2}\|\mathbf{V}-\mathbf{Y}\|_{fro}^2 + \frac{1}{2}\|\mathbf{U}-\mathbf{X}\|_{fro}^2,$$
$$\textit{subject to} : \mathbf{V} = \mathbf{AU}$$

where $\mathbf{Y}, \mathbf{V} \in \mathbb{R}^{n\times k}, \mathbf{X}, \mathbf{U} \in \mathbb{R}^{d\times k}$. The solution is given by:

$$\mathbf{U} = [\mathbf{A}^T\mathbf{A} + \mathbf{I}]^{-1}(\mathbf{X} + \mathbf{A}^T\mathbf{Y}) \tag{2.3}$$
$$\mathbf{V} = \mathbf{AU} \tag{2.4}$$

The above computation is preferred when $d \ll n$. When d is larger, the solution may be rewritten in terms of an $n \times n$ linear system involving $\mathbf{AA}^T$ instead (see [13]). However, the $d \ll n$ case is more relevant for our setting.

To adapt this approach to our setting, we assume an implicit logical block partitioning of the matrix $\mathbf{Z}$ into $R \times C$ blocks, where R and C denote row and column splitting parameters respectively.

$$\begin{pmatrix} \mathbf{X}_1 \\ \vdots \\ \mathbf{X}_R \end{pmatrix} \begin{matrix} \rightarrow \\ \vdots \\ \rightarrow \end{matrix} \begin{pmatrix} \mathbf{Z}_{11} & \mathbf{Z}_{12} & \dots & \mathbf{Z}_{1C} \\ \vdots & \vdots & \vdots & \vdots \\ \mathbf{Z}_{R1} & \mathbf{Z}_{R2} & \dots & \mathbf{Z}_{RC} \end{pmatrix}, \begin{pmatrix} \mathbf{Y}_1 \\ \vdots \\ \mathbf{Y}_R \end{pmatrix} \tag{2.5}$$

We assume that $\mathbf{X}_i, \mathbf{Y}_i, \mathbf{Z}_{ij}$ have n_i rows where $\sum_{i=1}^{R} n_i = n$ while $\mathbf{Z}_{ij}$ has s_j columns where $\sum_{j=1}^{C} s_j = s$. We assume that R parallel MPI processes are invoked on the N nodes, each spawning $t < T$ threads that collectively own the parallel computation related to the

C blocks of $\mathbf{Z}_{ij}, j = 1 \dots C$. Each $\mathbf{Z}_{ij}$ is a function of the corresponding $\mathbf{X}_i$, and hence we exploit shared memory parallelism for computations across the column blocks of $\mathbf{Z}$. Thus, each of the C column blocks is assigned to one of the t threads, and blocks assigned to a single thread are processed sequentially.

To interpret the block-splitting ADMM algorithm, it is convenient to setup the following semantics. Let $\mathbf{W}_{ij} \in \mathbb{R}^{s_j \times m}$ denote local model parameters associated with block $\mathbf{Z}_{ij}$. We require each local model to agree with the corresponding block of global parameters, i.e., $\mathbf{W}_{ij} = \mathbf{W}_j$. The partial output of the local model on the block $\mathbf{Z}_{ij}$ is given by $\mathbf{O}_{ij} = \mathbf{Z}_{ij}\mathbf{W}_{ij} \in \mathbb{R}^{n_i \times m}$. The aggregate output across all the columns is $\mathbf{O}_i = \sum_{j=1}^{C} \mathbf{O}_{ij} = (\mathbf{ZW})_i$.

Let the set of n_i indices in the i^{th} row block be denoted by I_i. We denote the local loss measured by i^{th} MPI process as,

$$l_i(\mathbf{O}_i) = \frac{1}{n} \sum_{j \in I_i} V(\mathbf{y}_j, \mathbf{o}_j), i = 1 \dots M$$

where $\mathbf{o}_j^T, j \in I_i$ are rows of the matrix $\mathbf{O}_i$. Similarly, we assume that the regularizer r in (1.2) is separable over row blocks, i.e. $r(\mathbf{W}) = \sum_{j=1}^{C} r_j(\mathbf{W}_j)$ where $\mathbf{W}_j \in \mathbb{R}^{s_j \times m}$ is the conforming block of rows of $\mathbf{W}$. This assumption holds for l_2 regularization, i.e., $r_j(\mathbf{W}_j) = \|\mathbf{W}_j\|_{fro}^2$.

With the notation setup above, it is easy to see that Eqn. 1.2 can be equivalently rewritten over blocks as follows,

$$\begin{aligned} \underset{\mathbf{W} \in \mathbb{R}^{s \times k}}{\operatorname{argmin}} & \sum_{i=1}^{R} l_i(\mathbf{O}_i) + \lambda \sum_{j=1}^{C} r_j(\mathbf{W}_j) && (2.6) \\ & + \sum_{i,j} \mathbb{I}_{\mathbf{Z}_{ij}}(\mathbf{O}_{ij}, \mathbf{W}_{ij}) \\ \text{subject to} \quad & \mathcal{C}_1 : \mathbf{W}_{ij} = \mathbf{W}_j, && (2.7) \\ & \mathcal{C}_2 : \mathbf{O}_i = \sum_{j=1}^{N} \mathbf{O}_{ij} && (2.8) \end{aligned}$$

for $i = 1 \dots R, j = 1 \dots C$, with $\mathbb{I}_{\mathbf{Z}_{ij}}(\mathbf{O}_i^j, \mathbf{W}_j^i) = 0$ if $\mathbf{O}_i^j = \mathbf{Z}_{ij}\mathbf{W}_j^i$ and ∞ otherwise. Viewed as a convex constrained optimization problem, one can follow the derivation of block-splitting ADMM [13] by introducing new consensus variables $\overline{\mathbf{W}}_j, \overline{\mathbf{W}}_{ij}, \overline{\mathbf{O}}_i, \overline{\mathbf{O}}_{ij}$ corresponding to $\mathbf{W}_j, \mathbf{W}_{ij}, \mathbf{O}_i, \mathbf{O}_{ij}$ and associated dual variables $\boldsymbol{\mu}_j, \boldsymbol{\mu}_{ij}, \boldsymbol{\nu}_i, \boldsymbol{\nu}_{ij}$. Furthermore, the projection onto the constraint sets $\mathcal{C}_1$ and $\mathcal{C}_2$ turn out to have closed form averaging and exchange solutions. [13] note that $\boldsymbol{\nu}_{ij}$ can be eliminated since $\boldsymbol{\nu}_{ij}$ turns out to equal $-\boldsymbol{\nu}_i$ after the first iteration. Similarly, $\overline{\mathbf{W}}_{ij} = \overline{\mathbf{W}}_j$ and hence $\overline{\mathbf{W}}_{ij}$ can also be eliminated. These simplifications imply the final modified update equations derived in

[13]:

$$\begin{aligned}
\mathbf{O}_i &= \texttt{prox}_{\frac{1}{\rho} l_i}\left[\overline{\mathbf{O}}_i - \boldsymbol{\nu}_i\right] && (2.9)\\
\mathbf{W}_j &= \texttt{prox}_{\frac{1}{\rho} r_j}\left[\overline{\mathbf{W}}_j - \boldsymbol{\mu}_j\right] && (2.10)\\
(\mathbf{O}_{ij}, \mathbf{W}_{ij}) &= \texttt{proj}_{\mathbf{Z}_{ij}}[\overline{\mathbf{O}}_{ij} + \boldsymbol{\nu}_i, \overline{\mathbf{W}}_j - \boldsymbol{\mu}_{ij}] && (2.11)\\
\overline{\mathbf{W}}_j &= \frac{1}{R+1}\left(\mathbf{W}_j + \sum_{i=1}^{R} \mathbf{W}_{ij}\right) && (2.12)\\
\overline{\mathbf{O}}_{ij} &= \mathbf{O}_{ij} + \frac{1}{C+1}\left(\mathbf{O}_i - \sum_{j=1}^{C} \mathbf{O}_{ij}\right) && (2.13)\\
\overline{\mathbf{O}}_i &= \sum_j \overline{\mathbf{O}}_{ij} && (2.14)\\
\boldsymbol{\mu}_j &= \boldsymbol{\mu}_j + \mathbf{W}_j - \overline{\mathbf{W}}_j && (2.15)\\
\boldsymbol{\mu}_{ij} &= \boldsymbol{\mu}_{ij} + \mathbf{W}_{ij} - \overline{\mathbf{W}}_j && (2.16)\\
\boldsymbol{\nu}_i &= \boldsymbol{\nu}_i + \mathbf{O}_i - \overline{\mathbf{O}}_i && (2.17)
\end{aligned}$$

where i runs from 1 to R and j from 1 to C.

2.1 Modularity

Note that the loss function and the regularizer only enter the ADMM updates via their proximal operator. Thus a user needs to only specify a sequential proximal operator function, for the ADMM wrapper to immediately yield a parallel solver. While our current implementation supports squared loss, l_1 loss, hinge loss and multinomial logistic loss, our experiments will focus on the hinge loss case that corresponds to the support vector machine (SVM) model. The proximal operator for the hinge loss has a closed form solution [12].

2.2 Generating Random Transforms On-the-Fly

We assume that the block ADMM procedure is given a transform operator $\mathbb{T}$ which when applied to $\mathbf{X}_i$ given a block id j, produces the output $\mathbf{Z}_{ij}$, i.e.,

$$\mathbb{T}[\mathbf{X}_i, j] = \mathbf{Z}_{ij}$$

This transform function is used to generate $\mathbf{Z}_{ij}$ as needed in the optimization process, used and discarded in each iteration. The construction of these transform operators will be discussed in section 3

2.3 Caching

The graph projection step (2.11) requires the computation of (2.3) with $\mathbf{A} = \mathbf{Z}_{ij}$, i.e.,

$$\begin{aligned}
\mathbf{W}_{ij} &= \mathbf{Q}_{ij}[\overline{\mathbf{W}}_j - \boldsymbol{\mu}_{ij} + \mathbf{Z}_{ij}^T(\overline{\mathbf{O}}_{ij} + \boldsymbol{\nu}_i)] && (2.18)\\
\mathbf{O}_{ij} &= \mathbf{Z}_{ij}\mathbf{W}_{ij} && (2.19)
\end{aligned}$$

where

$$\mathbf{Q}_{ij} = [\mathbf{Z}_{ij}^T\mathbf{Z}_{ij} + \mathbf{I}]^{-1}. \quad (2.20)$$

The matrix $\mathbf{Q}_{ij}$ (or the Cholesky factors of the inverse above) can be cached during the first iteration and reused for faster solves in subsequent iterations. The cache requires

$O(\sum_{j=1}^{C} s_j^2)$ memory, or assuming that $s_j = \frac{s}{C}$, $O(\frac{s^2}{C})$ memory. Thus increasing the column splitting reduces the memory footprint. It also reduces the total number of floating-point operations required for (2.18), to $O(\frac{s^2}{C})$.

2.4 Memory Requirements, Variable Elimination and Parallel Execution

In the form presented in [13], the block splitting ADMM algorithm is not scalable in our setting despite the high degree of parallelism in it. This is because a naive implementation of the algorithm requires each node/process to hold the C local matrices $\mathbf{O}_{ij}, \overline{\mathbf{O}}_{ij}$ for a total memory requirement which grows as $2n_iCm$. This can be quite substantial for moderate to large values of the product Cm since n_i is expected to still be large. As an example, if $C = 64$ is set as the number of threads on BG/Q, for a 100-class classification problem, the maximum number of examples that a node can hold before the entire 16-GB memory is consumed by just one of these variables alone, is barely 335000. The presence of these variables conflicts with the need to increase C to reduce the memory requirements and increase parallelism for solving the Graph projection steps in (2.18),(2.19). Fortunately, the materialization of these variables can also be avoided as follows, by noting the form of the solution of Graph projection and exploiting shared memory access of variables across column blocks.

First, the variable $\overline{\mathbf{O}}_{ij}$ only contributes to a running sum in (2.14) and appears in the Graph projection step (2.11), (2.18) in the context of the product $\mathbf{Z}_{ij}^T\overline{\mathbf{O}}_{ij}$. These steps, together with the update of $\overline{\mathbf{O}}_{ij}$ in (2.13) can be merged while eliminating each of the C variables, $\overline{\mathbf{O}}_{ij}$, as follows. We introduce an $s \times k$ variable $\mathbf{U}_i \in \mathbb{R}^{s\times m}$ and instead maintain $\mathbf{U}_{ij} = \mathbf{Z}_{ij}^T\mathbf{O}_{ij} \in \mathbb{R}^{s_j\times m}$. A single new variable $\Delta \in \mathbb{R}^{n_i\times m}$ tracks the value of $\mathbf{O}_i - \sum_{j=1}^{C} \mathbf{O}_{ij}$ which is updated incrementally as $\mathbf{O}_{ij}$ is generated from (2.19). Eqn.2.13 implies the following update,

$$\mathbf{Z}_{ij}^T\overline{\mathbf{O}}_{ij} = \mathbf{U}_{ij} + \frac{1}{C+1}\mathbf{Z}_{ij}^T\Delta$$

which can be used in (2.18). The update in (2.14) can also be replaced with $\overline{\mathbf{O}}_i = \frac{1}{C+1}\sum_{j=1}^{C} \mathbf{O}_{ij} + \frac{C}{C+1}\mathbf{O}_i$. Thus the updates can be reorganized much more efficiently with no more than one (expensive) call to the random features transform function $\mathbb{T}$. These modified update equations are given in Algorithm 2.1, which is what is executed by an MPI process with t OMP threads. Note that since the loss term l_i is separable over the n_i data points, step A can be parallelized over multiple threads. Similarly for step B involving the prox operator computation with respect to the regularization function r.

Algorithm 2.1: BLOCKADMM$(i, \mathbf{X}_i, \mathbf{Y}_i, l_i, r, \mathbb{T})$

Initialize : $\mathbf{O}, \overline{\mathbf{O}}, \boldsymbol{\nu}, \bar{\Delta} = 0 \in \mathbb{R}^{n_i \times m}$,
$\overline{\mathbf{W}}, \mathbf{W}'\boldsymbol{\mu}', \boldsymbol{\mu}, \mathbf{U} = 0 \in \mathbb{R}^{s \times m}$
if $i = 0$, **Initialize** : $\mathbf{W} = 0 \in \mathbb{R}^{s \times m}$
for iter $= 1 \ldots$ max
A. $\mathbf{O} = \texttt{prox}_{\frac{1}{\rho} l_i}\left[\overline{\mathbf{O}} - \boldsymbol{\nu}\right]$ (1)
if $i = 0$:
B. $\mathbf{W}_j = \texttt{prox}_{\frac{1}{\rho} r_j}\left[\overline{\mathbf{W}}_j - \boldsymbol{\mu}_j\right], j = 1 \ldots C$
C. BROADCAST$(\overline{\mathbf{W}})$
$\Delta = \mathbf{O}$
$\overline{\mathbf{O}} = \frac{C}{C+1}\mathbf{O}$
// thread t executes the following
D. **for** $j = \frac{(t-1)C}{T} + 1 \ldots \frac{tC}{T}$
E. $\mathbf{Z}_{ij} = \mathbb{T}[\mathbf{X}_i, j]$
if $iter = 0$, setup cache: $\mathbf{Q}_{ij}$ (2.20)
F. $\mathbf{A} = \frac{1}{C+1}\mathbf{Z}_{ij}^T \bar{\Delta}$
G. $\mathbf{W}'_j = \mathbf{Q}_{ij}[\overline{\mathbf{W}}_j - \mu'_j + \mathbf{U}_j + \mathbf{A}]$
H. $\mathbf{O}' = \mathbf{Z}_{ij}\mathbf{W}'_j$
$\Delta = \Delta - \mathbf{O}'$ // synchronized
I. $\mathbf{U}_j = \mathbf{Z}_{ij}^T \mathbf{O}'$
$\overline{\mathbf{O}} = \overline{\mathbf{O}} + \frac{1}{C+1}\mathbf{O}'$ // synchronized
$\boldsymbol{\mu}'_j = \boldsymbol{\mu}'_j + \mathbf{W}'_j - \overline{\mathbf{W}}_j$
$\bar{\Delta} = \Delta$
if $i = 0$:
J. $\boldsymbol{\mu} = \boldsymbol{\mu} + \mathbf{W} - \overline{\mathbf{W}}$
K. $\overline{\mathbf{W}} = \frac{1}{R+1}$REDUCE$(\mathbf{W}')$
L. $\overline{\mathbf{W}} = \overline{\mathbf{W}} + \frac{1}{R+1}\mathbf{W}$
M. $\boldsymbol{\nu} = \boldsymbol{\nu} + \mathbf{O} - \overline{\mathbf{O}}$

Memory Requirements: Assuming $s_j = \frac{s}{C}$, the total memory requirements per node can be computed as follows:

$$4n_i m + 5sm + n_i d + n_i m + \frac{Tn_i s}{C} + \frac{sm}{C} + n_i m T + \frac{s^2}{C}$$

where the terms can be associated with the variables $(\mathbf{O}, \overline{\mathbf{O}}, \boldsymbol{\nu}, \bar{\Delta})$, $\overline{\mathbf{W}}, \mathbf{W}'\boldsymbol{\mu}', \boldsymbol{\mu}, \mathbf{U}$; the data $\mathbf{X}_i, \mathbf{Y}_i$; the materialization of the block $\mathbf{Z}_{ij}$ across the T threads; temporary variables to implement step D; and the factorization cache. Thus increasing the column splitting C and reducing the number of threads T provide knobs with which to satisfy memory constraints.

Computational Complexity: In terms of computation, the prox operator computation in steps A and B parallelize over multiple threads and have linear complexity. The three dominant computational phases are:

- Cost of invoking the transform $\mathbb{T}[\mathbf{X}_i]$ in Step E, which tends to be the cost of right matrix multiplication of a random $d \times s_j$ matrix against $\mathbf{X}_i$, i.e. $O(\frac{ndS}{TCN})$. For certain transforms this can be accelerated as discussed in section 3.
- $O(\frac{s^2 m}{TC})$ cost of Graph Projection step G.
- $O(\frac{nsm}{TN})$ Cost of matrix multiplications against blocks of $\mathbf{Z}$ in steps F,H and I.

Communication: The cost of broadcast and reduce operations in step C and K costs grow as $O(s\ m\ log\ N)$.

3. Randomized Kernel Maps

We now discuss the role of the random feature transforms $\mathbf{z}(\cdot)$ in our distributed solver. Since the initial work of Rahimi and Recht, several alternative mappings have been suggested in the literature. These different mappings tradeoff the complexity of the transformation on dense/sparse vectors, kernel approximation, kernel choice, memory requirements, etc. Our algorithm essentially operates on $\mathbf{z}_1, \ldots, \mathbf{z}_n$ which are defined by $\mathbf{z}_i = z(\mathbf{x}_i)$. So, once we have selected a $\mathbf{z}(\cdot)$ it can be treated as a black-box as all other steps of the algorithm are the same. This gives a lot of flexability to our algorithm to operate on different kernels, and different kernel maps. The choice of $\mathbf{z}(\cdot)$ also encapsulates different treatment for sparse or dense input – as all maps produce dense $\mathbf{z}$s, different treatment of sparse and dense input appears solely in the application of $\mathbf{z}(\cdot)$.

Our block splitting scheme uses at each point of time, on a given node, only some of the $\mathbf{z}$s, and in each one of them, only a part of the vector. The former is easily addressed by applying $z(\cdot)$ only to the $\mathbf{x}$s of interest. To achieve the latter, given a known scheme for generating kernel maps, we construct our kernel map $z(\cdot)$ as follows

$$z(\mathbf{x}) = \frac{1}{\sqrt{s}} \left[\sqrt{s_1} z_1(\mathbf{x}) \ldots \sqrt{s_C} z_C(\mathbf{x})\right]^T$$

where $z_j : \mathbb{R}^d \mapsto \mathbb{R}^{s_j},\ \ j = 1, \ldots, C$ is a feature map generated independently. That is, we use of Monte-Carlo approximation. We can then use z_j to generate blocks $\mathbf{Z}_{1j}, \mathbf{Z}_{2j}, \ldots, \mathbf{Z}_{Rj}$ for $j = 1, \ldots, C$.

We now describe the various kernel mapping we implemented in our solver, and the implementation issues that arise. In the description we describe how to construct a $z : \mathbb{R}^d \mapsto \mathbb{R}^s$ with the intention that this scheme is used to compute $z_1(\cdot), \ldots, z_C(\cdot)$ as explained above.

- **Random Fourier Features:** This the mapping suggested by Rahimi and Recht [15]. It is designed for the Gaussian kernel $k(\mathbf{x}, \mathbf{z}) = \exp(-\|\mathbf{x} - \mathbf{z}\|_2^2 / 2\sigma^2)$ (for some $\sigma \in \mathbb{R}$) [1] The mapping is $z(\mathbf{x}) = \cos(\omega^T \mathbf{x} + \mathbf{b})/\sqrt{s}$ where $\omega \in \mathbb{R}^{d \times s}$ is drawn from an appropriately scaled Gaussian distribution, $\mathbf{b} \in \mathbb{R}^s$ is drawn from a uniform distribution on $[0, \pi)$, and the cosine function is applied entry-wise. For dense input $\mathbf{x}$ is $T_z(\mathbf{x}) = O(sd)$, and for sparse input $\mathbf{x}$ is $T_z(\mathbf{x}) = O(s\,\mathtt{nnz}(\mathbf{x}))$. When applied to a group of inputs collected inside a matrix (as in generating $\mathbf{Z}_{ij}$ from $\mathbf{X}_{ij}$), most of the operations can be done inside a single GEMM, which gives access to highly tuned parallel BLAS implementations.

  Notice that naively representing $z(\cdot)$ on a machine requires $O(sd)$ memory (storing the entries in ω), which can be rather costly. We avoid this by keeping an implicit representation in terms of state of the pseudo random number generator, and generating parts of the ω on the fly as needed.

- **Fast Fast Random Fourier Features:** Also called Fastfood in [19], this approach targets the same class of kernels and uses a similar scheme as Random Fourier Features. An accelerated approach is used for generating ω as a multiplication of random diagonal matrices, permutation matrices, and FFT matrices. As such, the vector $\omega^T \mathbf{x}$

[1] The construction suggested by Rahimi and Recht actually spans a full family of shift-invariant kernels.

can be computed much faster in $O(s \log d)$ time. Sparsity cannot be exploited to yield faster running time, as can be done for random Fourier features. In terms of approximation quality, it is slightly worse than random Fourier features, but not in a significant manner.

The use of the widely used FFT operation allows access to highly tuned implementations of that operation. On the flip side, matrix-matrix operations (i.e. FFT on the columns/rows of a matrix) are not as well tuned and effective in the use of the cache hierarchy as level-3 BLAS operations.

Extensions of these constructions for other classes of kernels have also been proposed, e.g. the Random Laplace Features of [20] for non-negative domains, and TensorSketch method of [21] for approximating the polynomial kernel.

4. Experimental Evaluation

Datasets: We report experimental results on two widely used machine learning datasets: `MNIST` (image classification) and `TIMIT` (speech recognition). `MNIST` is a 10-class digit recognition problem with training set comprising of $n = 8.1M$ examples and a test set comprising of $10K$ instances. There are $d = 784$ features derived from intensities of 28×28 pixel images. `TIMIT` is 147-class phoneme classification problem with a training set comprising of $2,251,569$ examples and a test set comprising of 115934 instances. The input dimensionality is $d = 440$.

Cluster Configuration: We report our results on two distributed memory computing environments: a BlueGene/Q rack ($1,024$ nodes, 16-cores per node and 4-way hyperthreading), and a 20-node commodity cluster TRILOKA with 8 cores per node. The latter is representative of typical cloud-like distributed environments, while the former is representative traditional high-end HPC machines.

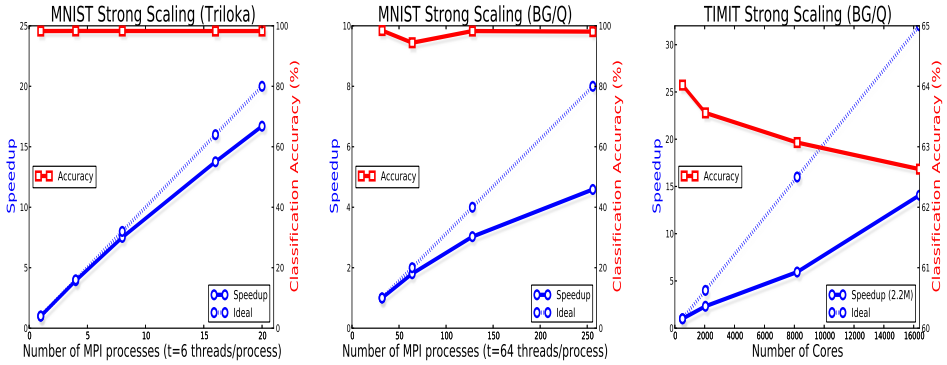
Default parameters and Metrics: Unless otherwise mentioned, we report results with Gaussian kernels and Hinge Loss (SVMs). We use Random Fourier Feature maps, and store the input dataset using a dense matrix representations. We report speedups, running times for fixed number of ADMM iterations, and classification accuracy obtained on a test set.

4.1 Strong Scaling Efficiency

Evaluating strong scaling in our setup should be approached with caution. The standard notion of strong scaling is to take a fixed problem size and study the speedup obtained as the number of processing elements are increased. It is generally assumed that parallelization accelerates a sequential algorithm, but does not change too much (or at all) the results and their quality. Thus, the focus is on the computational gains and communication overhead tradeoffs.

While it possible to fix the amount of work in our algorithm by fixing n, d and the number iterations, we are not guaranteed to provide same quality results as the number of row split increase (i.e., as we use more processors). ADMM guarantees only *asymptotic* convergence to the same solution, irrespective of data splitting. This introduces statistical tradeoffs since in practice, machine learning algorithms rarely attempt to find very high-precision solutions to the optimization problem, since the goal is to estimate a model that generalizes well, rather than solve an optimization problem (indeed, it can be rigorously argued that optimization error need not be reduced below statistical estimation errors [22]). Increased row splitting implies that ADMM coordinates among a larger number of local

Figure 4.1: Strong-scaling experiments.



models, each of which is statistically weaker, so we expect some slow down of learning rate in a strong scaling regime. Thus, our experiments also evaluate how row splitting affects the classification accuracy.

Results of our strong scaling experiments are shown in Figure 4.1.

MNIST: The number of random features is set to $100K$, and we use 200 column partitions. On TRILOKA we use $n = 200K$ while varying the number of processors from 1 to 20. We observe nearly ideal speedup. On BG/Q we use $n = 250K$, and vary the number of nodes from 32 to 256. We measure speedup and parallel efficiency with respect to 32 nodes. We observe nearly ideal speedup on 64 nodes. With higher node counts the parallel efficiency start to decline, but it is still pretty good (57%) on 256 nodes. The increased row splitting causes non-significant slowdown in learning rate

TIMIT: We use the entire dataset and experiment only on BG/Q. The number of random features is set to $176K$, and we use 200 column partitions. Speedup is not far from linear, and parallel efficiency is 40% for 256 nodes. In terms of learning rate, accuracy curve declines and the slowdown is apparent, implying that more iterations are required to yield similar quality model.

4.2 Weak Scaling Efficiency

The weak scaling regime of interest is one in which the number of random features and number of iterations stay constant, and the number of examples grows with the number of processors[2]. Contrary to a strong scaling regime, in such a regime we expect the classification accuracy to increase as more parallel resources are pulled in. In fact, this regime is more interest as the main purpose of our solver to allow scalable learning on many examples.

Results are reported in Figure 4.2. On TRILOKA , number of examples is increased at the rate of $10K$ examples per node. On BG/Q (`MNIST` only), the number of examples is increased at the rate of $250K$ per node. Results show nearly constant running time. In terms of improvement in classification accuracy, we see significant improvement of models trained on TRILOKA as the number of examples increase. The gains are modest for BG/Q runs since the baseline model trained on $250K$ examples already has high accuracy.

[2]Although we caution that there are non-trivial interaction between the various parameters, so in fact, one may want to increase the number of random features and number of iterations when more examples are used. However, exploring these interactions is not the scope of this article

Figure 4.2: Weak Scaling experiments

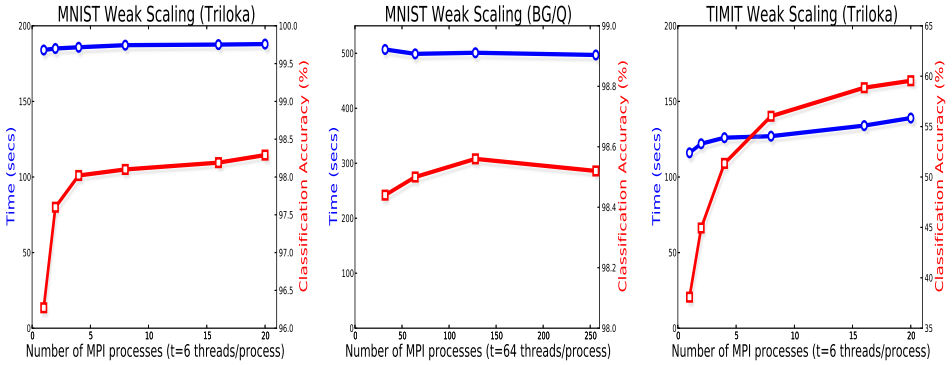
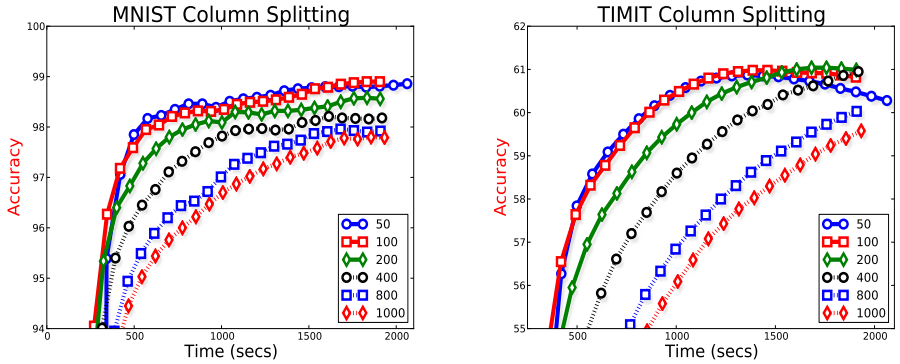


Figure 4.3: Effect of Column Splitting



4.3 Effect of Column Partitions

Figure 4.3 shows the effect of increasing the number of column splits, C, when using $100K$ random features for both `MNIST` and `TIMIT`. Increasing C reduces memory requirements and improves the running time of Graph projections. This comes with the tradeoff that the rate of learning per iteration can be significantly slowed, e.g. with $C = 1000$. At the same time, the plot shows that $C = 50, 100, 200$ perform similarly, and hence the optimization admits lower memory execution with little loss in terms of quality of the results.

As a rule of thumb, we advocate setting C to roughly s/d, since that ensures the ability to accommodate the memory requirements as long as the input matrix does not use more than $1/t$ of the available memory.

4.4 Performance Breakdown for training big models

Table 4.4 shows the performance breakdown of a model learnt on the full `TIMIT` dataset, with $528K$ random features on BG/Q with 256 nodes. This model returns state of the art accuracy (see Figure 1.1). Note that if we materialize $\mathbf{Z}$ explicitly, 8.7-terabytes of memory will be required. As Table 4.4 shows, the Graph projection loop, in particular the feature transformation step and the matrix multiplication against blocks of $\mathbf{Z}$, dominate the running time (which includes prediction on the test set at every iteration). The proximal operator costs is minimal because of closed form solutions that is embarrassingly parallel over multiple threads. The communication costs are also non-significant for reducing and broadcasting model parameters which in this case are buffers of size 0.6 GB. To reduce the overall time per iteration further, we plan to investigate a stochastic version of the algorithm

where only a random subset of blocks are updated.

Step	Min Time	Max Time	Avg Time	Percentage
Communication (C,K)	81	396	381	5%
Transform (E in D)	2175	2185	2180	29.4%
Graph Projection Loop (D)	6464	6548	6512	88%
Proximal Operators (A,B)	0.19	0.21	0.20	0%
Barrier	0	55	54	0.7%
Prediction	417	501	453	6%
Total	7419	7419	7419	100%

4.5 Comparison against Sequential and Parallel Solvers

Here we provide a sense of how our solvers compare with two other solvers. The first one, LibSVM [23][3], is widely acknowledged to be the state-of-the-art sequential solver. The other one, PSVM [24], is an open source MPI-based parallel SVM code. PSVM computes in parallel a low-rank factorization of the Gram matrix via Incomplete Cholesky and then uses this factorization to accelerate a primal-dual interior point method for solving the SVM problem.

We compiled a multithreaded version of LibSVM. Since PSVM only supports binary classification, we created versions of `MNIST` and `TIMIT` by dividing their classes into a positive and negative class. A comparison is shown below for an SVM problem with Gaussian kernels ($\sigma = 10$) and regularization parameter $\lambda = 0.001$. We use 20 TRILOKA nodes with 6 cores for PSVM and our solver. LibSVM, with its default optimization parameters, requires more than a day to solve the binary `MNIST` problem and about 22 hours to solve the binary `TIMIT` problem. The testing time is also significant particularly for `TIMIT` which has a large test set with more than $115K$ examples. The main reason for this slow prediction speed is that the number of support vectors found by LibSVM is not small: $15,667$ for `MNIST` and $45,071$ for `TIMIT`, making the evaluation of the kernel expansion computationally expensive. This problem is shared by PSVM though its prediction speed is faster due to parallel evaluation. However, PSVM runtimes rapidly increase with its rank parameter p. For $p = \sqrt{n}$, advocated by the accompanying paper [24], the estimated model provides a much worse accuracy-time tradeoff than our solver, which can work with much larger low-rank approximations that are computed locally and cheaply. On both datasets, our solver approaches the LibSVM classification accuracy.

Table 1: Comparison on MNIST-binary (200k)

	Libsvm	PSVM ($n^{0.5}$)	BlockADMM
Training Time	108720	194.15	178.82
Testing Time	169	8.45	1.63
Accuracy	98.51%	72.14%	97.55%

5. Conclusion

Our goal in this paper has been to resolve scalability challenges associated with kernel methods by using randomization in conjunction with distributed optimization. We noted

[3] `http://www.csie.ntu.edu.tw/~cjlin/libsvm/`

Table 2: Comparison on TIMIT-binary (100k)

	Libsvm	PSVM ($n^{0.5}$)	BlockADMM
Training Time	80355	47	42
Testing Time	1295	259	2.9
Accuracy	85.41%	73.1%	83.47%

that this combination leads to a class of problems involving very large implicit datasets. To handle such datasets in distributed memory computing environments where we also want to exploit shared memory parallelism, we investigate a block-splitting variant of the ADMM algorithm which is reorganized and adapted for our specific setting. Our approach is high-performance both in terms of scalability as well as in terms of statistical accuracy-time tradeoffs. The implementation supports various loss functions and is highly modular. We plan to investigate a stochastic version of our approach where only a random selection of blocks are updated in each iteration. We also plan to investigate methods to sketch the data so as to improve the memory requirements of our algorithm with respect to number of output coordinates.

Acknowledgements

The authors acknowledge the support from XDATA program of the Defense Advanced Research Projects Agency (DARPA), administered through Air Force Research Laboratory contract FA8750- 12-C-0323.

References

[1] G. Hinton, L. Deng, D. Yu, G. Dahl, A. rahman Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. Sainath, and B. Kingsbury, "Deep neural networks for acoustic modeling in speech recognition," *Signal Processing Magazine*, 2012.

[2] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf, "Deepface: Closing the gap to human-level performance in face verification," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2014.

[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," 2012.

[4] R. Socher, A. Perelygin, J. Wu, J. Chuang, C. Manning, A. Ng, and C. Potts, "Recursive deep models for semantic compositionality over a sentiment treebank," in *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2013.

[5] P.-S. Huang, H. Avron, T. Sainath, V. Sindhwani, and B. Ramabhadran, "Kernel methods match deep neural networks on timit," *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, 2014.

[6] B. Schlkopf and A. J. Smola, *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. MIT Press.

[7] L. Song, B. Boots, S. Siddiqi, G. Gordon, and A. Smola, "Hilbert space embeddings of hidden markov models," in *International Conference in Machine Learning (ICML)*, 2013.

[8] B. Boots, A. Gretton, and G. J. Gordon, "Hilbert space embeddings of predictive state representations." in *Uncertainty in Artificial Intelligence (UAI)*, 2013.

[9] K.-B. M. for Hypothesis Testing: A Unified View, "Z. harchaoui and f. bach and o. cappe and e. moulines," 2013.

[10] K. Zhang, J. Peters, D. Janzing, and B. Scholkopf, "Kernel based conditional independence test and application in causal discovery," in *Uncertainty in Artificial Intelligence (UAI)*, 2011.

[11] L. Bottou, D. D. Olivier Chapelle, and J. W. (Editors), *Large-scale Kernel Machines*. MIT Press.

[12] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein, "Distributed optimization and statistical learning via the alternating direction method of multipliers," *Foundations and Trends in Machine Learning.*, vol. 3, no. 1, pp. 1–122, 2011.

[13] N. Parikh and S. Boyd, "Block splitting for distributed optimization," *Math. Prog. Comp.*, October 2013.

[14] G. Wahba, *Spline Models for Observational Data*. CBMS-NSF Regional Conference Series in Applied Mathematics.

[15] A. Rahimi and B. Recht, "Random features for large-scale kernel machines," in *Neural Information Processing Systems (NIPS)*, 2007.

[16] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou, "Communication-optimal parallel and sequential qr and lu factorizations," *SIAM J. Sci. Comput.*, vol. 34, no. 1, pp. 206–239, Feb. 2012. [Online]. Available: http://dx.doi.org/10.1137/080731992

[17] H. Avron, P. Maymounkov, and S. Toledo, "Blendenpik: Supercharging lapack's least-squares solver," *SIAM J. Sci. Comput.*, vol. 32, no. 3, pp. 1217–1236, Apr. 2010. [Online]. Available: http://dx.doi.org/10.1137/090767911

[18] X. Meng, M. A. Saunders, and M. W. Mahoney, "Lsrn: A parallel iterative solver for strongly over- or under-determined systems," *CoRR*, vol. abs/1109.5981, 2011.

[19] Q. Le, T. Sarlós, and A. Smola, "Fastfood – Approximating kernel expansions in loglinear time," in *International Conference on Machine Learning (ICML)*, 2013.

[20] J. Yang, V. Sindhwani, Q. Fan, H. Avron, and M. Mahoney, "Random laplace feature maps for semigroup kernels on histograms," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2014.

[21] N. Pham and R. Pagh, "Fast and scalable polynomial kernels via explicit feature maps," in *KDD*, 2013, pp. 239–247.

[22] L. Bottou and O. Bousquet, "The tradeoffs of large-scale learning," in *NIPS*, 2007.

[23] C.-C. Chang and C.-J. Lin, "Libsvm : a library for support vector machines." in *ACM Transactions on Intelligent Systems and Technology*, 2011.

[24] E. Y. Chang, K. Zhu, H. Wang, H. Bai, J. Li, Z. Qiu, and H. Cui, "Psvm: Parallelizing support vector machines on distributed computers," in *NIPS*, 2007, software available at http://code.google.com/p/psvm.