

UNIVERSITY COLLEGE LONDON

---

# Scaling Kernel SVM towards Big Data

A distributable Kernel SVM solver

---

Author:

ALEX BIRD

Supervisors:

JOHN SHAWE-TAYLOR

DAVID MARTÍNEZ-REGO

A thesis presented for the degree of  
MSc Computational Statistics and Machine Learning

7th September 2015

*This report is submitted as part requirement for the MSc Degree in Computational Statistics and Machine Learning at University College London. It is substantially the result of my own work except where explicitly indicated in the text.*

*The report may be freely copied and distributed provided the source is explicitly acknowledged.*



# Scaling Kernel SVM towards Big Data

by

Alex Bird

Submitted to the Department of Computer Science  
on Sept 7, 2015, in partial fulfillment of the  
requirements for the degree of  
MSc Computational Statistics and Machine Learning

## Abstract

Kernel SVM is a powerful learning algorithm, but few options are available in the data laden environment. We present a novel algorithm for SVM classification (LPSVM) based on LP boosting. This mechanism results in increasingly accurate sparse approximations to the SVM solution. The algorithm has a number of desirable qualities pertaining to distributed optimisation. These include low iteration count, controllable complexity and communication and the ability to operate on data in situ without pre-processing or re-distribution. The project includes a serial (single core) implementation which is competitive with many current serial approaches, and we discuss how the algorithm can be parallelised and the tasks for which it is best suited.

Code available at: <https://github.com/spoonbill/lpsvm>



## Acknowledgments

I would like to thank John Shawe-Taylor for his guidance, generosity with time and unwavering enthusiasm for the project. I would also like to extend my gratitude to David Martínez-Rego for many useful and enjoyable conversations, and most importantly, unhindered access to his ideas and initial drafts of the algorithm.



# Contents

<b>1</b>	<b>Introduction</b>	<b>9</b>
1.1	Support Vector Classification . . . . .	9
1.2	SVM for big data . . . . .	10
1.3	LPSVM . . . . .	11
1.4	Thesis structure . . . . .	11
<b>2</b>	<b>Background</b>	<b>13</b>
2.1	Single core kernel SVM solvers . . . . .	13
2.2	Distributed kernel SVM solvers . . . . .	15
2.3	Discussion . . . . .	16
<b>3</b>	<b>Previous Work</b>	<b>19</b>
3.1	LPBoost . . . . .	19
3.1.1	LPBoost motivation . . . . .	19
3.1.2	Column generation . . . . .	20
3.1.3	Reformulating the LP . . . . .	21
3.2	LPBoost for SVM training . . . . .	22
3.2.1	Boosting as margin maximisation . . . . .	22
3.2.2	Equivalence of LPBoost and SVC . . . . .	22
3.2.3	Optimal column generation . . . . .	24
3.2.4	Approximation of duality gap . . . . .	25
<b>4</b>	<b>LPSVM</b>	<b>27</b>
4.1	LPSVM overview . . . . .	27
4.1.1	Algorithmic details . . . . .	29
4.1.2	Generating the active set . . . . .	30
4.1.3	A parallel implementation . . . . .	32
4.1.4	Discussion . . . . .	32

<b>5 Experiments</b>	<b>35</b>
5.1 Datasets . . . . .	35
5.2 Algorithms . . . . .	36
5.3 Results . . . . .	36
5.4 Discussion . . . . .	38
5.5 Summary . . . . .	39
<b>6 Alternative optimisation</b>	<b>41</b>
<b>7 Conclusion</b>	<b>43</b>
<b>Bibliography</b>	<b>45</b>
<b>A Further algorithmic details for LPSVM</b>	<b>49</b>
A.1 KKT system for recovering primal variables . . . . .	49
A.1.1 KKT system for LPSVM dual . . . . .	50
A.1.2 Numerical precision . . . . .	52
A.1.3 Sparsity of weak learners . . . . .	54
A.2 Active set growth by iteration . . . . .	54
<b>B Details of alternative optimisation methods</b>	<b>59</b>
B.1 Introduction to penalty and augmented lagrangian methods . . . . .	59
B.2 Alternating Direction Method of Multipliers . . . . .	63
B.2.1 ADMM for Linear Programming . . . . .	65
B.2.2 Consensus ADMM for Linear Programming . . . . .	66
B.2.3 Consensus ADMM by Proximal Evaluation . . . . .	67
B.2.4 ADMM experiments . . . . .	69



# Chapter 1

## Introduction

Support Vector Machines are among the most important classification algorithms today and are used in a wide variety of fields and applications - image recognition, natural language processing, forecasting, genomics, medical imaging, environmental science, online advertising, e-learning, and many more. The advent of ‘Big Data’ would only seem to increase this, as large amounts of data become commonplace and new possibilities open up for science and industry alike. However, it also poses unique challenges for machine learning. While it seems obvious that more data should improve performance, poor scaling of algorithms can sometimes result in worse results than before. The purpose of this thesis is to explore using SVMs in a big data context and propose a novel algorithmic framework for doing so.

### 1.1 Support Vector Classification

Since there are many good introductions available for support vector machines (SVMs), such as [Cristianini and Shawe-Taylor, 2000], [Burges, 1998], we omit one here. However, to facilitate a common language, we seek to minimise the following optimisation objective:

$$\begin{aligned} \min_{w,b,\xi} \quad & \frac{1}{2} \|w\|^2 + C \sum_{i=1}^m \xi_i \\ \text{s.t.} \quad & y_i (\langle w, x_i \rangle + b) + \xi_i \geq 1 \quad i = 1, \dots, m \\ & \xi_i \geq 0 \quad i = 1, \dots, m \end{aligned} \tag{1.1}$$

where  $w$  is normal to the separating hyperplane,  $b$  is the bias, and  $\xi_i$  the margin violations. We will retain the convention of a training set  $\mathcal{S} = \{x_i, y_i\}_{i=1}^m$  with  $m$  training examples  $x_i \in \mathbb{R}^d$ , and labels  $y_i \in \{-1, +1\}$  throughout. A powerful aspect of SVM algorithms is the ability to map examples  $x_i$  into a high dimensional (possibly infinite) feature space without ever computing the map explicitly. This is possible due to the ‘kernel trick’ (see [Cristianini

and Shawe-Taylor, 2000] for more details). We are primarily interested in the kernelised version of SVM in this thesis due to its superior performance.

## 1.2 SVM for big data

Scaling kernel SVM to use in big data applications is a difficult problem. In this setting, the data is distributed across a cluster of nodes and is assumed large enough that it cannot fit in memory on a single machine. Distributed optimisation is usually challenging, but the optimisation involved in kernel SVMs has 3 distinct problems. Firstly, most *sequential* optimisers require  $O(m^2)$  time, and while there have been many attempts to reduce the complexity in the single core literature it remains an open problem. Quadratic time is even more impractical in the data laden setup. Secondly, the number of support vectors,  $n_{sv}$ , usually scales linearly with the size of the dataset [Steinwart, 2003]. This is problematic for a few reasons: *communication* - every support vector must be broadcast to every node, *memory* requirements are  $\Omega(n_{sv}^2)$ , and *test time* complexity is  $O(n_{sv})$ . The last point here is non-trivial since big data applications are frequently time critical, such as online advertising or image recognition. Finally, most SVM solvers are highly iterative in nature, that is, operations are performed example-by-example. Distributed optimisation will be extremely slow for such schemes due to the communication overhead and synchronicity.

We summarise some of these desirable qualities in the below, which will be referred to later:

1. **Low complexity.** Ideally the solver should compute the solution with a complexity that is sub-quadratic in  $m$  and independent of  $d$ . However a reasonable compromise may be that the time can be explicitly traded off with accuracy.
2. **Sparse solution.** The number of support vectors grows sublinearly with the dataset size, or can be controlled by the user. This is to reduce communication, memory and test time.
3. **Low communication.** While support vectors must be available to every node, all other data transfer must be kept to a minimum.
4. **Low iteration count.** We would prefer the algorithm to work on commodity hardware (eg. using map-Reduce) rather than requiring a dedicated MPI solution.
5. **Load balancing.** Resource wastage should be minimised; we do not want some nodes to be idle for significant periods.

It is unlikely that any solver can satisfy all of these qualities and inevitably some will be violated in favour of others.

## 1.3 LPSVM

Our approach to SVM, the *linear program SVM* (LPSVM) is a novel one within the SVM literature. Instead of directly solving the SVM objective, LPSVM attacks a boosting objective, which is constructed to be equivalent to an SVM solution at the optimum. Therefore, the optimisation pathway looks very different to other solvers. By imposing sparsity on the boosting algorithm, the optimisation iterates can be thought to provide a “budget” path to the SVM solution, where in each iteration, the classifier performs optimally<sup>1</sup> for a given support set size. These iterates inherit a number of attractive properties from boosting and can be used as sparse approximations to the SVM solution.

LPSVM is particularly suited to distributed environments, and is quite different in approach to other distributed solvers, discussed in chapter 2. By using boosting, the solver only needs to see a subset of the variables at a time. The master node or *fusion center* can perform the iterative optimisation, while the worker nodes communicate the variables as requested. For sparse solutions, the majority of datapoints in a node need not be communicated at all. The scheme fulfills some of the desirable qualities in section 1.2. It is linear in the number of datapoints, provided we fix the number of boosting rounds. The sparsity of each weak learner can be controlled, and so the overall sparsity can be upper bounded. Communication is limited provided the solution is sparse and the overall iteration count is the same as the number of boosting rounds, typically 100 – 200. However, we note that by using the fusion center set-up, worker nodes do waste resource during optimisation.

We implement a serial version LPSVM and the results suggest that LPSVM is a competitive solver, and it may be particularly useful for high dimensional, low noise datasets. While building a distributed implementation is out of scope for this project, we believe our results justify actively exploring such a project. We claim that LPSVM is a viable alternative to the current set of tools, and with some extra work could fulfill an important role in the distributed classifier toolbox.

## 1.4 Thesis structure

The rest of this thesis is structured as follows. Chapter 2 reviews the various classes of SVM solvers used today along with some of the distributed approaches in the literature. Chapter 3 summarises the previous work on LPBoost and its connection with SVM. Chapter 4 is a detailed description of LPSVM, and some preliminary results are shared in chapter 5. We

---

<sup>1</sup>with respect to the *boosting* objective

discuss alternative optimisation subroutines in chapter 6 and conclude the thesis in chapter 7. The appendices give some further technicalities for implementing the LPSVM procedure and also discuss the various optimisation approaches from chapter 6 in detail.

## Chapter 2

# Background

SVM research is extensive and ongoing. However, there is a notable scarcity of distributed solvers proposed. We will look first at an overview of the single core literature and then consider extensions toward distributed computation. Linear SVM solvers are omitted in the interest of focus.

### 2.1 Single core kernel SVM solvers

In this section we review the main classes of approach within SVM solvers. Distributed solvers either borrow from these ideas or explain why they are infeasible in each case. The number of approaches reflect the various trade-offs of speed and accuracy deemed appropriate for different applications.

**Mature exact solvers.** The first methods solved the dual QP exactly, addressing two problems with generic QP solvers: efficiency and memory limitations. Decomposition methods were among the earliest attempts to combat memory limitations, where at each iteration only a subset of indices are optimised. The most famous contributions include SMO [Platt et al., 1999] which optimises only two indices per iteration (coordinate descent), and shrinking [Joachims, 1999] which eliminates indices unlikely to change. These ideas still underlie state-of-the-art exact solvers such as LIBSVM [Chang and Lin, 2011] and *SVMlight* [Joachims, 1999]. These approaches work well and can be guaranteed to find an  $\epsilon$ -approximate solution in finite time [Lin, 2006], [Palagi and Sciandrone, 2005], but are impractical for datasets of  $O(10^6)$  examples. For example, experiments run by [Hsieh et al., 2013] show LIBSVM took 3.5 days to converge on the 8 million examples of the 8m MNIST dataset.

**Approximate dual solvers.** The efficiency of dual solvers can be significantly increased

if the kernel matrix admits a good low rank approximation. [Williams and Seeger, 2001] and [Fine and Scheinberg, 2002] use the Nyström approximation, a quadrature approach for low rank approximation which avoids computing the SVD. [Rahimi and Recht, 2007] instead sample random fourier features to learn an approximation to the feature map, and the resulting low dimensional space is classified using linear SVM. While significant gains can be made in speed, these approximations come at a cost of a  $O(1/\sqrt{m})$  worse generalisation error [Cortes et al., 2010],[Rahimi and Recht, 2009], where  $m$  is the number of samples used. [Yang et al., 2012] show that when there is a clear gap in the eigenspectrum, the Nyström method can outperform random features due to a reduced error bound of  $O(1/m)$ .

**Primal solvers.** Historically, SVM optimisation was approached via the dual program, but there is no particular reason to do so [Chapelle, 2007]. Indeed, Chapelle argues that *approximate* optimisation of the primal performs better than that of the dual. A number of primal solvers are available, currently the most well known is Pegasos [Shalev-Shwartz et al., 2011]. Pegasos optimises the primal using sub-gradient methods (SGD), and is shown to have a favourable  $\tilde{O}(1/\lambda\epsilon)$  convergence rate for linear problems with regularisation parameter  $\lambda$ , independent of training set size. Unfortunately this does not hold true for nonlinear kernels which converge as  $\tilde{O}(m/\lambda\epsilon)$ . Following this success of SGD, [Wang et al., 2012] propose a general framework for stochastic gradient solvers with a “budget” of support vectors. A variety of heuristics are investigated for maintaining the budget, and the authors report linear scaling for a given budget, which can be used for superior scaling of kernel Pegasos.

**Sparse solutions.** The linear growth in support set size is a long standing problem which was historically motivated by reducing prediction time at test stage. Reduced set methods such as [Burges et al., 1996], [Schölkopf et al., 1999], [Wu et al., 2005] act on the trained SVM solution to return a smaller set of “virtual vectors” that need not be part of the training set. A few solvers use heuristics to reduce the support size during training. For example, [Nguyen et al., 2010] find similar pairs of points belonging to the same class and combine them during training with decomposition methods. [Keerthi et al., 2006] propose a greedy method, *SpSVM*, which iteratively adds into the support set the example  $x_i$  which reduces the empirical risk the most. [Collobert et al., 2006] circumvent the linear growth in the support set by changing the objective. Here the hinge loss is replaced by the nonconvex ramp loss and optimised by the convex-concave procedure, with few observed problems relating to local minima.

**Recent developments.** There have been a few recent developments of large single core kernel SVM solvers, of which two are of particular interest. These algorithms were designed

and run on a single machine, but the ideas can be easily parallelised. The *low-rank Linearized SVM* (LLSVM) [Zhang et al., 2012] is another attempt to compute an empirical approximation of the kernel map. The approach here uses a factorisation of the Nyström method with a superior bound to random fourier features and unlike [Rahimi and Recht, 2007] is applicable to any kernel. Once the linear approximation is obtained, a distributed minimisation of the resulting SVM is performed, but on a single core. Secondly, the divide-and-conquer solver (DCSVM) [Hsieh et al., 2013] divides the problem into subproblems via a 2-stage kernel k-means application. Once the subproblems are solved, the solutions are “glued” back together and used to warm start global coordinate descent. Due to the division strategy, only a small number of global iterations are typically required. DCSVM outperforms LLSVM from experiments in [Hsieh et al., 2013], however, distributing the final co-ordinate descent stage of the former may be prohibitive in a map-Reduce environment.

## 2.2 Distributed kernel SVM solvers

There are notably few kernel SVM algorithms for distributed computing. Many of the ideas from large scale kernel solvers are highly iterative schemes and do not transfer readily into the distributed setting. We review many of the competitive ones below.

**Cascade-SVM** [Graf et al., 2004] was one of the earliest approaches for distributed SVM, based on filtering and combining support vectors from smaller subproblems. The data are randomly partitioned onto nodes which filter the support vectors from their subproblems. In the simplest arrangement, the network is arranged as a binary tree, and the support vectors found on level  $\ell - 1$  are recursively combined, re-solved and filtered on level  $\ell$ . The top level contains the optimal support vectors if and only if the support vectors are a subset of the filtered vectors from each subproblem. Any SVM solver may be used to solve the subproblems and the scheme can be iterated if required.

**Parallel SMO** [Cao et al., 2006] attacks the computationally expensive step in SMO: updating the gradient. The experiments were performed in an MPI context on a supercomputer, where 30 cores realised almost 70% the efficiency of the sequential solver. Recently [You et al., 2015b] demonstrate a number of highly optimised SVM HPC architectures which perform efficiently for different learning tasks. However, these ideas cannot be easily adapted to commodity hardware.

**PSVM** [Zhu et al., 2008] uses a parallel incomplete Cholesky factorisation to efficiently store the kernel matrix across nodes and then proceeds via a distributed interior point method. Interior point methods circumvent the communication problem, but are not particularly

competitive solvers. Nevertheless, the algorithm scales as  $O(m^2/k)$  for  $k$  workers up to a data dependent threshold (usually up to 100 nodes).

**P-packSVM** [Zhu et al., 2009] is a parallel stochastic gradient descent solver which runs in an MPI framework. The authors begin with the same SGD approach as Pegasos, but a distributed caching strategy is used to reduce kernel computations, and an innovative packing strategy used to condense many iterations into one. In the experiments, P-packSVM dominates PSVM in runtime and using 144 nodes, scores 99.49% accuracy in 3.5 hours on MNIST8m ( $8 \times 10^6$  examples, 784 features).

**map-Reduce TRON** [Mahajan et al., 2014] linearises with a similar method to LLSVM, but uses a trust region Newton method (TRON) to optimise the resulting objective. The reduction in iterations from using second order methods allows distributed optimisation in a Hadoop environment. The complexity scales as  $O(mp/k)$  for  $p$  Nyström basis points and  $k$  nodes, with order  $10^2$  iterations. The authors report increased accuracy and reduced runtime on commodity hardware versus the P-packsvm algorithm which employs an MPI cluster.

**CA-SVM** [You et al., 2015a] is a partitioning approach to the SVM problem which exploits fast decaying kernels such as  $\exp(1/\gamma\|x-v\|^2)$ . This method trains  $k$  different models on the  $k$  nodes based on the observation that the SVM prediction is only influenced by nearby examples. Hence the data are partitioned according to the dual objectives of euclidean-based clustering and load-balancing. The paper also demonstrates the poor scaling of a naive distributed DCSVM. Unlike Cascade SVM which communicates only the support vectors, DCSVM communicates each node’s entire dataset which is impractical as  $m$  grows large.

## 2.3 Discussion

We observe three broad categories in the above: adaptations of existing iterative algorithms in an HPC environment, second order approaches on commodity hardware, or heuristic divide-and-conquer. While DCSVM does not immediately work as a distributed algorithm, the door may yet be open to combinations with distributed optimisation such as P-packSVM. However, it is clear that the field of distributed SVM solvers is still in its infancy, and very few algorithms are available for commodity hardware.

LPSVM is clearly quite different from the approaches to date. map-Reduce TRON and Cascade-SVM appear to be the most competitive algorithms that can be used in Hadoop-like environments. Cascade-SVM can suffer from quite severe load-balancing problems, and cannot be performed in situ. Unfortunately we cannot easily evaluate map-Reduce TRON since not many experiments are available. Since LPSVM selects its basis points



optimally according to the boosting objective, and map-Reduce TRON selects its basis points arbitrarily, it may be that LPSVM can perform better. Nevertheless, due to the differences in the algorithms, one would at least expect the two to have a different set of strengths and weaknesses. The landscape of distributed kernel SVM solvers seems to be hospitable to new approaches.



## Chapter 3

# Previous Work

In this chapter we will see that the boosting algorithm LPBoost arises directly as a margin maximisation procedure. We will further demonstrate that the optimum of the boosting scheme is also an SVM solution under a restricted feature space. Finally we will look at a few practical aspects of LPBoost. Boosting itself is a powerful approach to create arbitrarily strong learners from an algorithm that generates weak learners. It is analogous to an adversary who consistently chooses the most difficult examples for the learner to classify. In doing so, the training error of the learning ensemble is driven to zero (under appropriate conditions). Since each boosting iteration also reduces a bound on the generalisation error, we expect a good classifier even if it is stopped early.

### 3.1 LPBoost

#### 3.1.1 LPBoost motivation

We present a brief overview of the motivation for LPBoost. This summary is intended only to emphasise the theoretical foundations of LPBoost, not to convey the details of the derivation. For an in-depth treatment refer to [Demiriz et al., 2002].

We consider the hypothesis class  $\mathcal{H}$  as the conical hull of an arbitrary set of weak learners:

$$co(H) = \left\{ \sum_{h \in H} a_h h : a_h \geq 0 \right\}$$

We assume  $H$  is closed under complementation and  $h$  are real-valued functions in  $[-1, +1]$ . By a covering number argument, Theorem 2.3 from [Demiriz et al., 2002] gives bounds on the generalisation error of a function  $f \in \mathcal{H}$  trained on a sample  $\mathcal{S}$  of  $\mathcal{D} := \mathcal{X} \times \{-1, +1\}$  with  $m$  examples. Let  $f$  achieve margin  $\gamma$  on  $\mathcal{S}$  in a feature space augmented by the slack

variables. With probability  $1 - \delta$  we have for the generalisation error  $\text{err}_{\mathcal{D}}(f)$ ,

$$\text{err}_{\mathcal{D}}(f) \leq \epsilon(m, \mathcal{H}, \delta, \gamma) = \frac{2}{m} \left( \log \mathcal{N}(\mathcal{G}, 2m \frac{\gamma}{2}) + \log \frac{2}{\delta} \right) \quad (3.1)$$

when  $m > 2/\epsilon$ . Here,  $\mathcal{N}$  denotes the covering number, and  $\mathcal{G}$  denotes the set  $\mathcal{G} := \{(\sum_{h \in H} a_h h, g) : \sum_{h \in H} a_h + \|g\|_1 \leq B, a_h \geq 0\}$ .  $g$  is proportional to the sum of the slack variables  $\xi$  associated with  $f$  on  $\mathcal{S}$ . Hence from  $\mathcal{G}$ , if

$$\sum_{h \in H} a_h + C \sum_{i=1}^m \xi_i \leq B$$

then we can apply bound (3.1). Further, we can bound the covering number  $\mathcal{N}(\mathcal{G}, 2m, \frac{\gamma}{2})$  from Theorem 2.4 in [Demiriz et al., 2002]

$$\log N(\mathcal{G}, 2m, \frac{\gamma}{2}) \leq 1 + \frac{144B^2}{\gamma^2} (2 + \ln(|\mathcal{G}| + m)) \log \left( 2 \left\lceil \frac{4B}{\gamma} + 2 \right\rceil m + 1 \right) \quad (3.2)$$

This analysis allows us to identify a good strategy for directly minimising the expected generalisation error. Suppose our set of weak learners  $h$  are the columns of a matrix  $H$ . Minimising bound (3.2) results in the following optimisation problem:

$$\begin{aligned} \min_{a, \xi} \quad & \sum_{i=1}^m a_i + C \sum_{i=1}^m \xi_i \\ \text{s.t.} \quad & y_i H_i a + \xi_i \geq 1, \quad \xi_i \geq 0, \quad i = 1, \dots, m \\ & a_j \geq 0, \quad i = 1, \dots, n \end{aligned} \quad (\text{LP } 0)$$

where the extra constraints arise from achieving a margin  $\gamma = 1$  chosen arbitrarily due to scale invariance in the preceding. The motivation above reveals an objective similar to SVM, and follows a similar argument. However, this objective permits an alternative optimisation route which we exploit in what follows.

### 3.1.2 Column generation

This problem (LP 0) is formulated as if all weak hypotheses were available and concatenated into the matrix  $H$ . LPBoost requires that we consider  $H$  to be the set of all possible labellings within the class  $\mathcal{H}$ . Of course even if the weak hypotheses only took values in the set  $\{-1, +1\}$ , the columns of  $H$  would have a cardinality exponential in  $m$ . Fortunately, the problem is still tractable since usually only a subset of the elements of  $a$  will be non-zero at the optimum. Further, if  $a$  is unconstrained, this subset need not be larger than  $m$ . We

index the non-zero subset by  $J' \subset J$ . If  $J'$  is known in advance, the problem will have the same solution restricted only to the learners  $J'$  as it does under all learners  $J$ .

Since we do not know the active constraint set in advance, we employ a heuristic for finding it. In *column generation* (CG) the original problem is known as the *master problem* (MP) and we solve the *restricted master problem* (RMP), which initially considers a small subset of columns and iteratively adds more until it contains the subset  $J'$ . Specifically in each iteration the RMP is solved and evaluated in the MP. If the solution is also feasible in the dual of the MP, our subset includes the active set and we have the optimal solution. Otherwise we obtain a constraint in the dual MP that is violated by the current solution and add this to the RMP. See [Desrosiers and Lübbecke, 2005] for a review of column generation in the context of integer programs<sup>1</sup>.

### 3.1.3 Reformulating the LP

Alternative formulations of (LP 0) exist. The following formulations from [Rätsch et al., 2000] and [Demiriz et al., 2002] are the primal and dual problems that we consider for LPSVM. The matrix  $H$  is specified by  $(H)_{ji} = y_i h_j(x_i)$ ; each row  $j$  corresponds to the margin of a hypothesis on the  $m$  examples<sup>2</sup>. Hence negative  $H_{ji}$  correspond to misclassification of hypothesis  $j$  on example  $i$ .

$$\begin{aligned}
\min_{a, \rho, \xi} \quad & D \sum_{i=1}^m \xi_i - \rho & (\text{P1}) \\
\text{s.t.} \quad & H^\top a + \xi \geq \rho \\
& \xi_i \geq 0, \quad i = 1, \dots, m \\
& \|a\|_1 = 1, \quad a_j \geq 0, \quad j = 1, \dots, n
\end{aligned}$$

$$\begin{aligned}
\max_{u, \beta} \quad & -\beta & (\text{D1}) \\
\text{s.t.} \quad & Hu \leq \beta \\
& \|u\|_1 = 1, \quad 0 \leq u_i \leq D, \quad i = 1, \dots, m
\end{aligned}$$

When  $n$  is finite,  $a \in \Delta_n$ , where  $\Delta_n$  is the  $n$ -dimensional simplex.

---

<sup>1</sup>column generation is frequently considered for ILPs since strong formulations of linear relaxation can involve an exponential number of columns. One might consider (LP 0) as a relaxed ILP where we look for a conic combination of learners created by subsamples of  $\mathcal{S}$  fed to the learning algorithm.

<sup>2</sup>this is the transpose of  $H$  as presented in the LPBoost paper

## 3.2 LPBoost for SVM training

We now show that under certain hypothesis classes, the solution to the master problem (P1) is equivalent to the solution to an SVM optimisation problem. The results in sections 3.2.2 and 3.2.3 are proved for the Multiple Kernel Learning problem in [Hussain and Shawe-Taylor, 2015]. We restate them as applied to the special case of SVMs.

### 3.2.1 Boosting as margin maximisation

Many boosting algorithms can be understood as margin maximisation algorithms. For instance, this is the most prominent explanation for why AdaBoost – probably the most widely known boosting algorithm – is resistant to overfitting. To be more precise, AdaBoost optimises the *margin distribution* rather than simply the margin, which may be a superior objective for some datasets. The difference between SVMs and AdaBoost lies in their respective margin definitions and that the SVM objective is  $\ell_2$  constrained, whereas AdaBoost is  $\ell_1$  constrained. For more details, the book [Schapire and Freund, 2012] is an excellent reference.

LPBoost, in contrast, optimises an upper bound of the AdaBoost cost function [Li and Shen, 2008]; LPBoost minimises the *max* of the margin, AdaBoost minimises a *log-sum-exp* of the data which is upper bounded by the max (up to an additive constant). The motivation behind LPBoost and SVM is sufficiently similar that it is not surprising that equivalence can be found. It is however unlikely that any other boosting algorithm will be useful in obtaining the SVM solution, although interesting algorithms may result in the attempt.

### 3.2.2 Equivalence of LPBoost and SVC

Define  $\mathcal{F}$  to be the *reproducing kernel Hilbert space* (RKHS) associated with the map  $k(\cdot, x) := \phi(x)$ , and let the hypothesis class  $\mathcal{H}$  be the unit ball in  $\mathcal{F}$ . Therefore

$$h(x) = \langle h, \phi(x) \rangle \tag{3.3}$$

for  $h \in \mathcal{F}$  such that  $\|h\| \leq 1$ . For this choice of hypothesis class, LPBoost can be shown to have the same solution as a SVC.

**Lemma 1** *The 1-Norm SVM has the same optimal solution as LP (P1) for hypothesis class  $\mathcal{H}$ . The LP formulation achieves this optimum when the weak learner  $h_1$  is the optimal SVC weight vector  $\mathbf{w}^*$  with weight  $a_1 = 1$ .*

### Proof

In [Cristianini and Shawe-Taylor, 2000] (chapter 7), the 1-norm SVC objective is given as:

$$\begin{aligned}
& \min_{\mathbf{w}, \rho, \xi} && D \sum_{i=1}^m \xi_i - \rho && (\text{SVC-P}) \\
& \text{s.t.} && y_i \langle \mathbf{w}, \phi(\mathbf{x}_i) \rangle \geq \rho - \xi_i, \quad i = 1, \dots, m \\
& && \|\mathbf{w}\|^2 = 1, \quad \xi_i \geq 0, \quad i = 1, \dots, m
\end{aligned}$$

Now relaxing the unit sphere restriction in (SVC-P) by expressing  $\mathbf{w}$  as the convex combination  $\sum_{h \in \mathcal{H}} a_h h$ , we have,

$$\begin{aligned}
& \min_{\mathbf{a}, \rho, \xi} && D \sum_{i=1}^m \xi_i - \rho && (3.4) \\
& \text{s.t.} && \sum_{h \in \mathcal{H}} y_i \langle \mathbf{h}, \phi(\mathbf{x}_i) \rangle a_h \geq \rho - \xi_i, \quad i = 1, \dots, m \\
& && \xi_i \geq 0, \quad i = 1, \dots, m \\
& && \|\mathbf{a}\|_1 = 1, \quad a_j \geq 0, \quad j = 1, \dots, n
\end{aligned}$$

which is equivalent to (P1). It remains to show that any convex combination  $\sum_{h \in \mathcal{H}} a_h h$  achieving less than unit norm is suboptimal.

First note that the optimum of (3.4) is nonpositive, since  $a_1 = 1, h_1 = \mathbf{0}$  is a feasible solution. Now suppose  $(a, \rho, \xi)$  is an optimal solution with associated  $w = \sum_{h \in \mathcal{H}} a_h h$  such that  $\|w\|^2 < 1$ . We have  $w/\|w\| \in \mathcal{H}$  and let  $e_w$  be its associated index vector. Then the solution  $(\hat{a}, \hat{\rho}, \hat{\xi}) := (e_w, \rho/\|w\|, \xi/\|w\|)$  is also feasible and has an objective value

$$\frac{1}{\|w\|} \left( D \sum_{i=1}^m \xi_i - \rho \right) < \left( D \sum_{i=1}^m \xi_i - \rho \right)$$

Thus  $(a, \rho, \xi)$  is not optimal. Hence by the choice of  $\mathcal{H}$ , the hypothesis of every optimal solution of (P1) lies on the unit sphere, proving the equivalence between (SVC-P) and (P1) under  $\mathcal{H}$ .

□

So access to a learning algorithm which can always generate  $h \in \mathcal{H}$  with training error  $< 1/2$  is sufficient for LPBoost to obtain the SVM solution on convergence. How do we generate hypotheses to make the greatest progress towards the optimum? For a given subset  $J'$  of the hypotheses, the solution of the RMP is always MP primal feasible, since it is the solution with the added constraint  $a_j = 0, \quad j \notin J'$ . If it is also MP dual feasible, we are

done since the duality gap is zero (see [Boyd and Vandenberghe, 2004]). If not, a constraint in (D1) will violate the minimax value  $\beta$ , and good progress can be made by adding the most violating of these into the RMP.

### 3.2.3 Optimal column generation

If an RMP solution  $(\hat{u}, \hat{\beta})$  violates the MP, we have  $\sum_{i=1}^m y_i h_j(x_i) \hat{u}_i > \beta$  for some  $j$ . We wish to maximise our progress identifying the required columns, so we look for the most violating hypothesis, ie.

$$\arg \max_{h \in \mathcal{H}} \sum_{i=1}^m y_i h(x_i) \hat{u}_i \quad (3.5)$$

Following the derivation in the multiple kernel learning setting from [Hussain and Shawe-Taylor, 2015], we have,

$$\begin{aligned} \arg \max_{h \in \mathcal{H}} \sum_{i=1}^m y_i h(x_i) \hat{u}_i &= \arg \max_{\|h\|_{\mathcal{F}} \leq 1} \sum_{i=1}^m \hat{u}_i y_i \langle h, \phi(x_i) \rangle \\ &= \arg \max_{\|h\|_{\mathcal{F}} \leq 1} \left\langle h, \sum_{i=1}^m \hat{u}_i y_i \phi(x_i) \right\rangle \end{aligned}$$

By the properties of the inner product, the maximum is attained at:

$$h = \frac{\sum_{i=1}^m \hat{u}_i y_i \phi(x_i)}{\left\| \sum_{i=1}^m \hat{u}_i y_i \phi(x_i) \right\|}$$

Let  $u^k$  be the optimal dual variable for the  $k^{\text{th}}$  CG iteration (henceforth ‘epoch’). Then the optimal weak learner to add to the RMP is

$$h_k = \frac{1}{v^k} \sum_{i=1}^m u_i^k y_i \kappa(x_i, \cdot) \quad \text{where} \quad v^k = \sqrt{\sum_{i=1}^m \sum_{j=1}^m u_i^k u_j^k y_i y_j \kappa(x_i, x_j)} \quad (3.6)$$

for the kernel function  $\kappa$  defined by the RKHS  $\mathcal{F}$ . Substituting this result into (3.5), the maximum achieved, and thus  $\min(H^{(k)} \hat{u})$  is  $v^k$ , so optimality can be tested via the condition

$$v^k \leq \hat{\beta}. \quad (3.7)$$

If (3.7) is true, then we have achieved the optimum, otherwise the new weak learner is added to the RMP and the algorithm continues.



### 3.2.4 Approximation of duality gap

The relationship between  $v^k$  and  $\hat{\beta}$  motivates a surrogate for monitoring the duality gap:  $\beta - v^k$  will tend to 0 and its magnitude may be informative of the iterates' suboptimality. We explore this idea more precisely. Consider that the dual of (SVC-P) is ([Cristianini and Shawe-Taylor, 2000], chapter 7):

$$\begin{aligned} \max_{\alpha} \quad & - \sqrt{\sum_{i,j=1}^m \alpha_i \alpha_j y_i y_j \kappa(x_i, x_j)} \\ \text{subject to} \quad & \|\alpha\|_1 = 1, \quad 0 \leq \alpha_i \leq D, \quad i = 1, \dots, m \end{aligned} \tag{SVC-D}$$

Each  $\hat{u}$  is feasible for (SVC-D) and so each  $-v^k$  is a lower bound on the SVM optimal value. Due to the equivalence of their optima in section 3.2.2, we also have that  $-v^k$  achieves the optimum on convergence. Likewise, by duality  $-\beta^k$  is the optimal value of the primal RMP, but since the RMP constitutes additional constraints, it is an upper bound for (P1) and moreover (SVC-P). Hence the interval  $[-v^k, -\beta^k]$  appears to be a good surrogate for the duality gap and will be tight on convergence.



# Chapter 4

## LPSVM

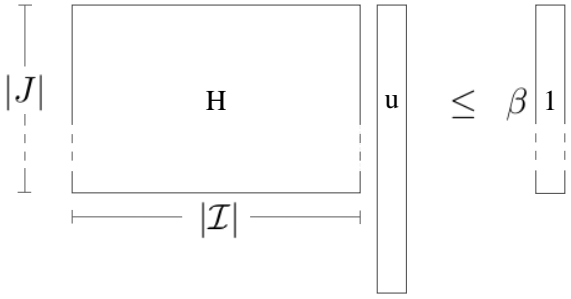
We now present the LPSVM algorithm, which is substantially based on the work in the previous chapter. Here we primarily address the practical aspects and how the algorithm relates to the project goals.

### 4.1 LPSVM overview

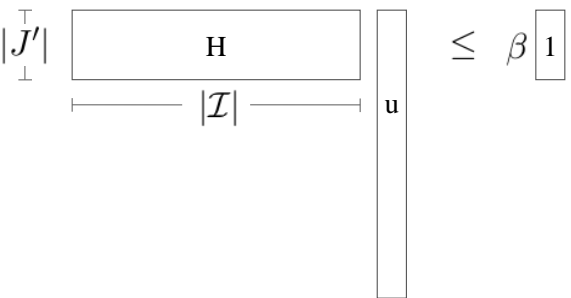
Chapter 3 introduces an algorithm for solving the SVM objective according to the framework of LPBoost. Specifically, we iterate solving a restricted version (RMP) of the LPBoost objective, which calculates the next weak learner and add it to the constraints in the RMP. This scheme is the application of column generation (CG) to the objective (P1), which results in an RMP with a large number of low dimensional constraints. Since the number of constraints is the same as the number of examples  $m$ , this may be infeasibly large in the big data context. It is therefore practical to apply CG to the dual of the RMP as well (see fig 4-1).

Formalising this idea, let  $\mathcal{I}$  be the index set of the examples, ie.  $|\mathcal{I}| = m$  and  $J$  be the index set of the weak learners  $h$ . Since we only ever operate on the second RMP (D-R2MP), the problem is of size  $|J'| \times |\mathcal{I}'|$ , where  $J' \subset J$  and  $\mathcal{I}' \subset \mathcal{I}$ . In practice this can always be made small enough to be solved by a single machine, since the size is explicitly controlled by the algorithm. Of course, if the SVM solution is dense in  $\mathcal{I}$ , we ultimately cannot control the size of the problem, and limiting the size of  $\mathcal{I}'$  may be detrimental to convergence. However, our intention is only to find good *sparse* solutions. If sparse solutions are unavailable then limiting the network traffic in a distributed SVM solve will be extremely challenging in any case, and may not even be desired due to slow evaluation at test time.

- The dual master problem:

$$\begin{aligned}
& \min_{u, \beta} \quad \beta & \text{(D-MP)} \\
& \text{s.t.} \quad \|u\|_1 = 1, \quad 0 \leq u_i \leq D, \quad i = 1, \dots, m,
\end{aligned}$$


- the dual of the restricted master program:

$$\begin{aligned}
& \min_{u, \beta} \quad \beta & \text{(D-RMP)} \\
& \text{s.t.} \quad \|u\|_1 = 1, \quad 0 \leq u_i \leq D, \quad i = 1, \dots, m,
\end{aligned}$$


- the restricted dual of the restricted master program:

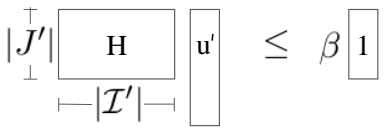
$$\begin{aligned}
& \min_{u, \beta} \quad \beta & \text{(D-R2MP)} \\
& \text{s.t.} \quad \|u\|_1 = 1, \quad 0 \leq u'_i \leq D, \quad i = 1, \dots, |\mathcal{I}'|,
\end{aligned}$$


Figure 4-1: The nested programs of LPSVM

---

**Algorithm 1** LPSVM (**args:** maxiter)

---

```
1: initialise  $H^{(1)}$ 
2: for epoch  $k = 1, \dots, \text{maxiter}$  do
3:   initialise  $\mathcal{I}'$ 
4:   for iteration  $i = 1, 2, \dots$  do
5:      $(\hat{u}, \hat{\beta}) := \text{solve D-R2MP}(H^{(k)}, \mathcal{I}')$ 
6:      $(\hat{a}, \hat{\rho}, \hat{\xi}) := \text{primal solution of D-R2MP}$ 
7:      $\delta := H^{(k)T} \hat{a} + \hat{\xi} - \hat{\rho}$  (primal violations )
8:     if  $\delta_i \geq 0$  for all  $i \in \mathcal{I}$  then
9:       set  $u^k := \hat{u}$ ,  $\beta^k := \hat{\beta}$ , break.
10:    else
11:      add subset of primal violations  $\{i : \delta_i < 0\}$  into  $\mathcal{I}'$ 
12:    end if
13:  end for
14:  calculate  $v^k$  for new weak learner.
15:  if  $v^k \leq \beta$  or  $k == \text{maxiter}$  then
16:     $a := \hat{a}$ ,  $\rho := \hat{\rho}$ , break
17:  else
18:    calculate  $h_k(x_i)$  for each  $i \in \mathcal{I}$ , and store as new row in matrix  $H^{(k+1)}$ 
19:  end if
20: end for
21: calculate support vector weights  $\alpha$  from  $\{u^j\}_{j=1}^k$  and weights  $a$ , return
```

---

The optimisation proceeds as follows. Let  $\text{D-R2MP}(H^{(k)}, \mathcal{I}')$  be the restriction of the dual MP to the constraint matrix  $H^{(k)}$  generated after the  $(k - 1)$ th epoch, with only columns  $\mathcal{I}'$  remaining. This restricted system is solved for  $(\hat{u}, \hat{\beta})$ . The primal variables can then be obtained either from the lagrange multipliers or the KKT system and tested for feasibility. If we have primal feasibility, both primal and dual problems are feasible with zero duality gap, and hence we have optimality. If not,  $\mathcal{I}'$  is increased to include some violating rows in the primal, and  $\text{D-R2MP}(H^{(k)}, \mathcal{I}')$  is solved again. Once optimality is achieved, the inner CG loop is complete. The dual variables found are used to obtain the next weak learner, which is the next CG epoch in the outer loop. The algorithm terminates when the approximate duality gap is below a given threshold (see chapter 3). This procedure is outlined in full in Algorithm 1.

#### 4.1.1 Algorithmic details

The theory underlying Algorithm 1 should be contained in chapter 3. At a high level, it is a simple scheme to solve a margin-maximising LP using column generation. The complications arise from making the solution practical. We address some practical concerns with the implementation below.

*Initialising  $H^{(1)}$ :* If we begin CG with a zero row matrix  $H^{(0)}$ , any value of  $u$  that satisfies the simplex  $\Delta_m$  and box constraints  $\mathcal{B}$  is feasible, and so provided  $\Delta_m \cap \mathcal{B}$  is nonempty, we have many choices. A natural possibility is  $u_i = 1/m$  for all  $i$ . However, the resulting weak learner will require the full kernel matrix to be calculated which should be avoided at all costs. A more useful choice might be to randomly select  $qm$  columns for a small choice  $q \ll 1$ , assigning each  $u_i$  equal weight. In the distributed setting, one may also wish to optimise the load balance across the nodes.

*Initialising  $\mathcal{I}'$ :* Naively this can be done in a similar way to initialising  $H^{(1)}$ . However, a few heuristics have proved useful in practice. By inspection of (D-R2MP), we should prefer columns whose projection along the  $-\mathbf{1}$  vector has a large magnitude. Ideally we must avoid selecting too many columns whose max elements align, but solving this exactly would be the solution to the LP and the approximate task is certainly non-trivial. We can also warm start the procedure by using the index set of the previous solution  $\{i : u_i > 0\}$ .

*Solving the LP:* There are a large number of good LP solvers available, both free and proprietary, open-source and not. Choice of solver is largely a matter of convenience and resource constraint. We have used the solvers from MATLAB’s Optimization Toolbox. Our experiments have shown that the *dual-simplex* method outperformed simplex and interior point methods in these implementations, but this may not be replicated in other libraries. Dual simplex methods are similar to the simplex method, but work with dual feasible rather than primal feasible iterates in the tableau. [Nocedal and Wright, 2006] claims that dual simplex methods are ‘often faster on many practical problems than [simplex methods]’.

*Solving for the primal variables:* It is counterintuitive to solve the dual problem only to use the primal variables. However, in our experiments the LP solver optimised the dual faster than the primal and scaled considerably better. The structure of the dual is simply a constrained minmax, which is a common LP problem. If the lagrange multipliers are available, the primal variables may be obtained immediately. If not (as is the case currently in MATLAB), one may solve the associated KKT system; we leave the details in the appendix.

#### 4.1.2 Generating the active set

Trading off iterations against the size of  $\mathcal{I}'$  (which we will term the *active set*) is a particular challenge. There is frequently no need to reduce the size of the active set with regard to the solver, since many solvers perform strategies similar to CG internally. However, we wish to reduce communication to a minimum. We investigate three strategies: *top-N*, *Gram-Schmidt* and the combination *GS+*

- *top-N*: The most obvious strategy. Fix a number  $N$  and for each active set iteration, the  $N$  columns with the largest constraint violations in the primal are added into  $\mathcal{I}'$ . If less than  $N$  columns violated the constraints then of course the lower number is used.
- *Gram-Schmidt* (GS): This strategy was motivated by the discrepancy between the number of columns added by *top-N* and the actual size of the active set. Gram-Schmidt was a strategy to reduce the number of unnecessary columns by iteratively removing (nearly) linearly dependent columns from the violating set. For details of the algorithm, see for example [Gentle, 2007]. Initialise by normalising the columns. The most violating column is added into  $\mathcal{I}'$  and its projection removed from the remaining set. Then we iterate by selecting at each stage the column with the largest remaining norm.
- *GS+*: The *GS* procedure is somewhat lacking since it does not consider the extent of the violation (except for the first column). *GS+* is a hybrid strategy which selects columns based on the multiplication of the violation and the remaining norm from GS.

## Experiments

The goal is to understand the number of CG iterations required to find the active set for each strategy. To benchmark, we fix  $N$  for each experiment; that is, each strategy adds the same number of columns in each iteration. For the Gram-Schmidt strategies, 80% of the projected value is iteratively taken from the remaining vectors rather than 100% to allow for meaningful selection when  $N$  is larger than the column dimension. Figure 4-2 shows the results for different values of  $N$  and dimension  $k$ . Results were averaged over 10 attempts on the MNIST dataset for the ‘round digit’ task (see chapter 5). The experiments were also replicated for a more sparse task with no substantial change shown in the results.

Gram-Schmidt performs worse than the top-N strategy on all results. One interpretation of this is that the optimal vertex of the feasible polytope is unlikely to be defined only by orthogonal constraints. As the iterates approach the optimum, one would expect the surface to be constrained by a number of similar datapoints which the GS strategy takes longer to find. Indeed, preliminary results indicate that on average, GS outperforms the top-N strategies for the first 2 iterates, but then converges much slower. However, the gains to be made by exploiting this are not large. Top-N and GS+ perform similarly, and for ease of implementation, Top-N is preferred.

For the size of dataset SVM is traditionally benchmarked on,  $N$  need not be selected with too much care - a constant value between 100 – 150 will suffice for  $D > 0.01$ . However,

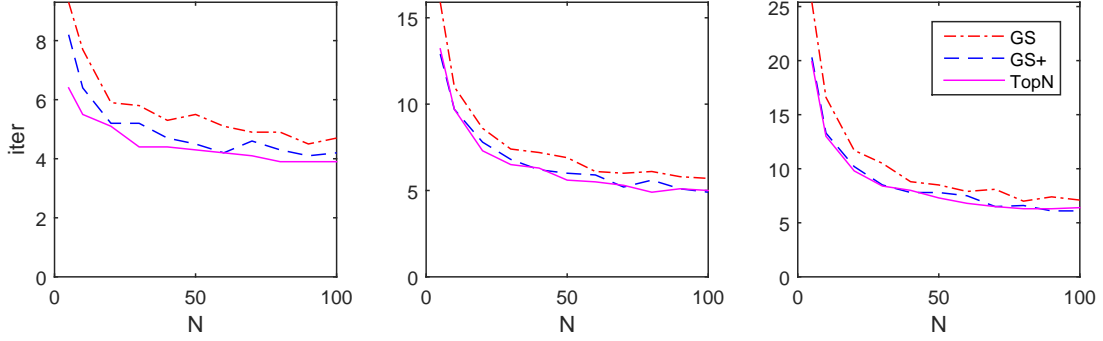


Figure 4-2: Iterations required for the active set strategies on MNIST. The plots show from *left to right* results for dimension  $k = 3, 10, 20$ .

reducing the communication required for a large candidate set size  $|\mathcal{I}'|$  will be necessary for the distributed implementation. We must assign costs to the number of iterations and  $|\mathcal{I}'|$  to optimise a good choice of  $N$ , which is not possible without performing experiments in a cluster. See appendix A.2 for more details of the (single node) experiments.

#### 4.1.3 A parallel implementation

The matrix  $H^{(k)}$  will be distributed column-wise over the network in a distributed setting. A basic implementation of LPSVM here is for each node to send the columns corresponding to  $\mathcal{I}'$  to the *fusion center* where the LP (D-R2MP) is solved on a single core. The resulting  $(\hat{a}, \hat{\rho})$  would be broadcast to the storage nodes, where either a subset of violating columns would be returned or a certificate of feasibility. Once the fusion centre has obtained certificates of feasibility from all nodes, it must gather the required examples  $\{i : u_i > 0\}$  from the storage nodes and compute  $v^k$ . This may be achieved either centrally or via an *all-to-all* broadcast. Finally the storage nodes compute the next row for  $H^{(k+1)}$  using the broadcast examples and  $v^k$ , and the scheme iterates. The most attractive part of the scheme is that all kernel computations take place in minibatches on each node.

#### 4.1.4 Discussion

We believe this is an attractive set-up for a distributed SVM solver with respect to the five qualities described in chapter 1.

- Complexity is not directly related to the number of examples. The number of examples required in the active set is a function of  $D$ , and the dimension of the problem is  $k$ , the number of weak learners generated thus far. However, the number of epochs required



for convergence may increase in the data-laden environment.

- Sparsity can be simply controlled in the model. The number of support vectors added in each epoch is at most  $\lceil 1/D \rceil + p$  (see appendix A.1.3). Using the  $\nu$ -SVC terminology [Rätsch et al., 1999],  $\frac{1}{D} = \nu m$ , which is the estimated upper bound the of margin violations in the data. Since all margin violations must be support vectors, it is quite natural to consider  $D$  as a sparsity inducing parameter. Large values of  $D$  yield sparse learners which can be computed quickly, whereas small values give rise to slower dense learners.
- Communication involves only the sparse set of examples at each epoch  $\{i : u_i > 0\}$  and the iterative gathering of columns in the active set stage. Either one of these stages may involve large amounts of data but they are controllable, rather than requiring constant iterative communication between nodes. For non-sparse solutions, this remains a concern.
- LPSVM is inherently a low iteration procedure. While a non-sparse problem may require a large number of iterations for a poor choice of  $D$ , this can easily be diagnosed by specifying a low value for `maxiter` initially. For problems which are limited by iterations, we can guarantee that the solution improves with more iterations since boosting is the optimisation mechanism.
- Worker nodes remain largely idle during the optimisation phase. When the kernel computations dominate the runtime, this is not a problem, but particularly for low dimensional data, resource is wasted through a single-core optimisation.

LPSVM will work best when the number of epochs  $k$  is kept to a minimum,  $D$  is large and kernel computations dominate. This means that it is ideal for high dimensional, sparse deterministic datasets. However, if we could implement a *distributed* solver, the algorithm will be performant for less sparse, and potentially more noisy datasets. We mention one drawback of LPSVM – we have no results on the number of iterations required for convergence to the SVM solution. However in practice good approximations are usually found quickly.



## Chapter 5

# Experiments

We evaluate the algorithm on three datasets commonly used to benchmark SVM algorithms. Building a distributed version of the LPSVM algorithm is out of scope for this project. Instead we demonstrate the viability on serial application and discuss the expected benefits of the distributed version. Since the algorithm usually converges slowly to the *exact* SVM solution, we report the progress and error at various intervals to examine the effect of early stopping.

### 5.1 Datasets

We used four datasets to benchmark the algorithm, [LeCun et al., 1998, Prokhorov, 2001, Webb et al., 2006, Krizhevsky and Hinton, 2009]. The parameters are shown in table 5.1. In the table,  $m$  and  $m_t$  are the training and test size respectively,  $d$  is the dimension of the dataset. We have chosen these datasets to capture variability over different  $m$  and  $p$  values. As in [Zhang et al., 2012], we have transformed MNIST into a binary classification task by labelling digits with straight edges (1,2,4,5,7) versus those without (3,6,8,9,0). The webspam dataset has been reduced in size due to poor memory management in our LPSVM implementation.

dataset	$m$	$d$	$m_t$	C	$\gamma$	D
MNIST	60,000	784	10,000	0.003	0.033	0.02
IJCNN	49,990	22	91,701	32	2	0.01
webspam*	125,000	254	50,000	8	10	0.01
CIFAR	50,000	3072	10,000	8	$4 \times 10^6$	0.01

Table 5.1: Dataset and parameter values

## 5.2 Algorithms

Our purpose is to show performance in accuracy and time over the convergence of our algorithm, so for the main comparison we use LIBSVM ([Chang and Lin, 2011] v3.20). No pre-processing steps are involved in either algorithm. We also report benchmarks from [Hsieh et al., 2013] which contain some of the strongest recent attempts for serial SVM. Distributed benchmarks are omitted, since it is difficult to make a meaningful comparison.

The parameters were chosen by cross validation where they differ from those in [Hsieh et al., 2013] and are shown in table 5.1. In all datasets we use the Gaussian RBF kernel,  $k(x, x') = \exp\{\frac{1}{\gamma}||x - x'||^2\}$  for convenience, but any kernel can be used in LPSVM. The regularisation parameter  $C$  in SVM and  $D$  in LPSVM do not have a one-to-one correspondence of (see [Demiriz et al., 2002]), so these were chosen independently, often balancing speed with accuracy. Our experiments are performed on a PC with Intel i5 3210M 2.50GHz processor and 8GB RAM, for which the experimental times prove faster than the benchmarks set in [Hsieh et al., 2013].

## 5.3 Results

Table 5.2 demonstrates the top level performance of serial LPSVM versus LIBSVM. Comparison with the benchmarks solvers is also shown where applicable in table 5.2. We have used results from early stopping of LPSVM, due to the efficiency drop approaching the optimum. Unfortunately the results for MNIST and webspam are incomparable with those from the DCSVM paper, so these are omitted. The algorithm looks to be competitive with other kernel SVM solvers – it can usually obtain an approximate solution in much less time than LIBSVM without significant loss in accuracy.

	MNIST		IJCNN		webspam125k		CIFAR	
	Time (s)	Acc(%)	Time (s)	Acc(%)	Time (s)	Acc(%)	Time (s)	Acc(%)
*LPSVM	687	98.49	74	97.92	579	98.62	9671	88.10
LIBSVM	1399	98.97	47	98.58	1601	98.99	29609	89.72
DCSVM			41	98.69			16314	89.50
DCSVM early			12	98.35			1977	87.02
CascadeSVM			17.1	98.08			6148	86.80
SpSVM			20	94.92			21335	85.60
LLSVM			38	98.23			9745	86.50

Table 5.2: LPSVM and benchmark results

A sampled trajectory of LPSVM can be found in table 5.3 – for the various datasets. We assume the kernel computations can be parallelised but the solver time to be irreducible,

	k	50	100	150	200	LIBSVM
MNIST	time (s)	295	687	1216	1909	1399
	–kernel	274	591	909	1154	–
	–solver	21	96	308	755	–
	nSVs	1808	3964	6072	7706	8956
	test acc %	97.36	98.49	98.73	98.86	98.97

	k	50	100	143*	–	LIBSVM
IJCNN	time (s)	22	74	224	0	47
	–kernel	6	10	12	–	–
	–solver	16	64	212	–	–
	nSVs	1252	1846	2209	–	2169
	test acc %	96.55	97.92	98.04	0.00	98.58

	k	20	40	60	73*	LIBSVM
IJCNN	time (s)	59	87	115	140	47
	–kernel	50	64	70	73	–
	–solver	8	23	46	67	–
	nSVs	6551	7714	8023	7387	2169
	test acc %	96.91	98.80	98.63	98.62	98.58

	k	50	100	150	200	LIBSVM
webspam	time (s)	321	579	1095	2290	1601
	–kernel	294	446	569	750	–
	–solver	27	132	526	1540	–
	nSVs	2874	4349	5526	6530	6725
	test acc %	96.39	98.62	98.88	98.97	98.99

	k	50	100	150	–	LIBSVM
CIFAR	time (s)	2961	6294	9671	–	29609
	–kernel	2939	6177	9247	–	–
	–solver	21	117	423	–	–
	nSVs	4838	10338	15738	–	31442
	test acc %	80.49	85.83	88.10	–	89.72

Table 5.3: LPSVM results for various suboptimal iterates (table headings). The elapsed time, split by kernel computations and solver time is shown, alongside the number of support vectors and the test error obtained from the current iterate. The results of LIBSVM at the optimum are provided for comparison. (\*) denotes convergence of LPSVM. The final sample is not available for CIFAR, but not due to algorithmic performance.

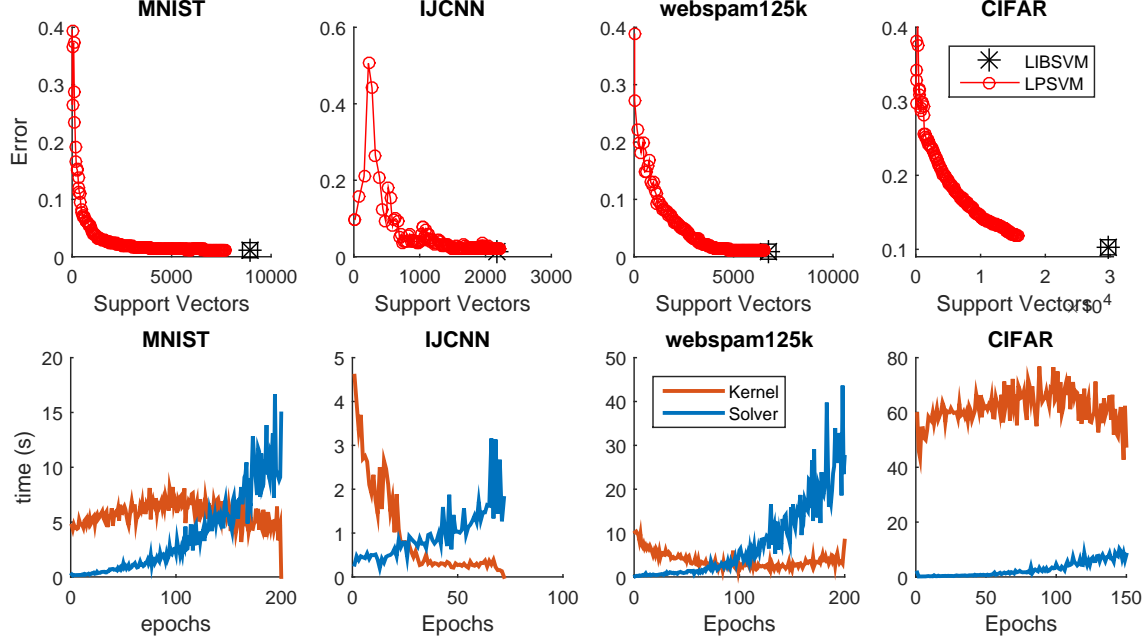


Figure 5-1: (top) Error rate (0-1) for support set size, (bottom) time per epoch for kernel computations and solver.

so these are broken out in the results. We believe there is good potential for parallelisation when the kernel computations dominate, as it does in high dimensional datasets such as MNIST or CIFAR. This is also true when the generated learners are non-sparse. However, as we approach 100 epochs, the solver time begins to dominate the runtime and parallelisation will not help, at least using the current solver. We may also infer from the IJCNN results that noisier data which requires lower margin penalties can not be optimised well using LPSVM. By using a lower value of  $D$ , and hence lower penalties, each weak learner is highly non-sparse (appendix A.1.3). Since the final hypothesis generated by LPSVM is a non-negative combination of these weak learners, the number of support vectors is likely to be many multiples of the number of non-zero  $u_i$  in each weak learner. Effectively, it overshoots, and there is sufficiently little to be gained from generating more sparse learners that it terminates with a very dense hypothesis.

## 5.4 Discussion

Figure 5-1 shows the convergence of the LPSVM solution to the LIBSVM solution as more support vectors are added. It is apparent that the LPSVM solution has a similar test error to LIBSVM when the number of support vectors are similar. Convergence to the same support set as LIBSVM is not particularly important, so there is no need to wait until the

optimum. The convergence also demonstrates a ‘diminishing returns’ effect for the number of support vectors. For instance, on the MNIST dataset, while LIBSVM has a test error of 1.03%, LPSVM can achieve 1.51% test error with fewer than half the support vectors. This might be considered a direct approach to the sparse SVM solutions encountered in chapter 2 which generally achieve sparsity as a post-processing step. While we do not claim that this is an optimally sparse SVM solution in the same sense, it may be a more practical alternative.

The approximation of the duality gap from chapter 3 is not always helpful in determining early stopping. The lower bound used is generally loose until close to the optimum, and so good performance can be obtained even when the gap appears large. In all of our experiments, stopping after 100 iterations (or if lower, on convergence) is a good compromise – these are the values used in table 5.2 for MNIST IJCNN and webspam125k. However, with a small amount of overhead, the exact dual value can be calculated from cached kernel values, or in a dedicated environment, one node could be used to calculate the cross-validation error online.

## 5.5 Summary

We can summarise these results as follows. LPSVM works well for more deterministic datasets such as images where margin penalties are large. It can also very effectively distribute the workload for very high dimensional data. However, in order to ensure that the algorithm works well for dense, low dimensional or noisy tasks, we might reasonably ask for: (i) separate parameters to control sparsity and regularisation, and (ii) a faster distributed solver. Nevertheless, the scheme can work well as is, and may find application in NLP, image recognition and bioinformatics.





## Chapter 6

# Alternative optimisation

We now turn to alternative methods of optimising the (D-R2MP) objective. The experiments show that as the number of epochs increase, the solver begins to dominate the runtime. While the approximations generated before this time may be adequate, for noisy or dense datasets, this will not always be true. Furthermore, we would like the computing power in the cluster to be used to its maximum potential. There may even be cases where the active set does not fit in memory, although given the low dimensional nature of the optimisation this is unlikely. Finally, there may be some gains to be made in reducing communication.

Despite the P-complete nature of the LP formulation, some parallel LP solvers exist, such as [Gondzio and Sarkissian, 2003]. We do not explore these further here, since the scope of our investigations is on single node architecture. Nevertheless, these may ultimately prove the most efficient way of achieving parallel efficiencies. Linear programs can not take advantage of much gradient information, and so are resistant to many standard machine learning tools. However, one would hope that by utilising the structure of the particular problem, we can do better than generic tools.

Unfortunately, our results from bespoke optimisation were negative, that is, the approach detailed in chapter 4 is superior. Thus they do not contribute to the ideas we present in the main text. Nevertheless, negative results are still results, and further details can be found in the appendix. We summarise the approaches taken briefly below.

Initially, projected first order methods such as stochastic subgradient methods and the Frank-Wolfe algorithm were considered. Results from initial implementations were not encouraging. Moreover, stochastic gradient methods are generally quick at finding approx-

imate solutions, but converge extremely slowly to high precision. In our case, precision does matter. As the LPSVM begins to converge, it becomes increasingly important to add the maximally violating weak learner. Suboptimal ones can make little difference to the objective at the cost of increasingly slow solver time. Once per epoch, we must also solve the KKT system to obtain the dual (primal) variables to check violation in the RMP.

The main body of the experiments were with augmented Lagrangian and ADMM methods (see appendix B). This area was investigated due to the convenient way that penalty methods handle constrained optimisation as well as recent experiments that suggest ADMM converges reasonably quickly for SVM. It is also a practical consideration, because many optimisation techniques fail entirely on linear programs. We note that like stochastic gradient methods, ADMM is also observed to converge slowly to high precision, but this is mostly from folklore than theoretical results.

A number of ADMM methods were implemented. These includes two explicitly distributed ideas – a consensus linear program method (LPSVM-D) and a fully distributed proximal evaluation scheme (LPSVM-PE) scheme. Both schemes exhibited significant oscillation in their convergence. As the number of virtual nodes in the implementation increased, the larger the oscillations became. This appears to be an artefact of the momentum in the control signal as the nodes fall into consensus. As well as slow convergence, oscillations mean that arbitrarily stopping the optimisation can result in a highly suboptimal result. The number of iterations required to obtain a good result numbered into the thousands for only moderately challenging tasks. We therefore conclude that this is unlikely to be a useful technique to improve the optimisation.

## Chapter 7

# Conclusion

We have been primarily looking to find a way to optimise kernel SVMs when the data are distributed in a commodity computing model. Very few approaches for this exist and it is not clear yet which is the best approach. Those that do exist have no way to ensure sparsity – a feature required for time critical prediction. In this thesis, we introduce LPSVM which is an entirely new approach to SVM optimisation. The algorithm is motivated by maximum margin theory and reaches the SVM objective via the route of an LPBoost ensemble. As a serial SVM solver it is competitive with many other recent approaches and in particular can compute faster and sparser solutions than LIBSVM. However, its main advantages lie in the potential to optimise the distributed SVM problem. The algorithm can operate on data in situ, avoids communicating all data which lies far from the margin and computes a sparse pathway to the SVM solution. We have discussed one possible distributed implementation and discussed tasks on which it may perform well.

We still believe there is scope for a good distributed optimisation algorithm to improve performance. This may allow a greater number of iterations to be performed before the solver bottleneck begins and make it a more accessible algorithm for dense problems. It may also reduce communication, since no fusion center is required. We have discussed a number of ADMM consensus implementations that were not able to realise the speedup or communication reduction required. However, other options remain, for instance parallel interior point solvers. Besides a distributed solver, separating the sparsity parameter from the regularisation parameter will be necessary for some datasets. While sparsity and regularisation are related, there must be some flexibility permitted in the model, and this currently precludes using the algorithm on noisy datasets.

These additions would make LPSVM a more attractive algorithm. Nevertheless, there is

certainly reason to believe that a distributed version of the current algorithm will be useful. For example, experiments on the CIFAR dataset spent over 95% of the time performing kernel computations. This is exactly the kind of situation in which LPSVM confers an advantage.

One final investigation may be required at this stage to understand the scaling of the algorithm to very large datasets. Currently this is not possible due to poor memory management. The only ways in which large volumes of data can affect the complexity is by increasing the candidate set size for finding the active set, and increasing the number of iterations required for convergence. Thus sub-quadratic complexity may be obtained, but we will require empirical proof.

There are a number of minor improvements which may improve the algorithm considerably. We do not yet have an intelligent heuristic for building the active set in the inner column generation loop. Given that the candidate set is usually many multiples of the active set size, communication can be drastically reduced. A heuristic analogous to shrinking may also help. The algorithm currently computes entire rows of the kernel matrix at a time. If an oracle revealed that some examples were not support vectors, kernel computations could be reduced.

A significant problem with the formulation as an LP is that the gradient is (necessarily) constant. This reduces the amount of information about examples considerably, meaning that a number of heuristics devised for the standard SVM problem cannot be used, including shrinking or working set selection. This includes directional information, hence the difficulties in finding better optimisation approaches. However, there are significant advantages in linear problems, and the resources of the most mature field in optimisation available for use.

We believe that LPSVM constitutes a framework that may be of significant use for large-scale SVM problems in its current form. Moreover, despite some negative results in further optimisations, it may be ripe for improvement by more experienced hands.

# Bibliography

- Dimitri P Bertsekas. *Constrained optimization and Lagrange multiplier methods*, volume 1. Athena Scientific, Belmont, MA, 1996.
- Léon Bottou and Chih-Jen Lin. Support vector machine solvers. *Large scale kernel machines*, pages 301–320, 2007.
- Stephen Boyd. Lecture slides: Alternating direction method of multipliers. [http://stanford.edu/class/ee364b/lectures/admm\\_slides.pdf](http://stanford.edu/class/ee364b/lectures/admm_slides.pdf), 2011.
- Stephen Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.
- Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, and Jonathan Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends® in Machine Learning*, 3(1):1–122, 2011.
- Christopher JC Burges. A tutorial on support vector machines for pattern recognition. *Data mining and knowledge discovery*, 2(2):121–167, 1998.
- Christopher JC Burges et al. Simplified support vector decision rules. In *ICML*, volume 96, pages 71–77. Citeseer, 1996.
- Li Juan Cao, S Sathiya Keerthi, Chong-Jin Ong, Jian Qiu Zhang, and Henry P Lee. Parallel sequential minimal optimization for the training of support vector machines. *Neural Networks, IEEE Transactions on*, 17(4):1039–1049, 2006.
- Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011. Software available at <http://www.csie.ntu.edu.tw/~cjlin/libsvm>.
- Olivier Chapelle. Training a support vector machine in the primal. *Neural Computation*, 19(5):1155–1178, 2007.
- Ronan Collobert, Fabian Sinz, Jason Weston, and Léon Bottou. Trading convexity for scalability. In *Proceedings of the 23rd international conference on Machine learning*, pages 201–208. ACM, 2006.
- Corinna Cortes, Mehryar Mohri, and Ameet Talwalkar. On the impact of kernel approximation on learning accuracy. In *International Conference on Artificial Intelligence and Statistics*, pages 113–120, 2010.
- Nello Cristianini and John Shawe-Taylor. *An introduction to support vector machines and other kernel-based learning methods*. Cambridge university press, 2000.

- Ayhan Demiriz, Kristin P Bennett, and John Shawe-Taylor. Linear programming boosting via column generation. *Machine Learning*, 46(1-3):225–254, 2002.
- Jacques Desrosiers and Marco E Lübbecke. *A primer in column generation*. Springer, 2005.
- John Duchi, Shai Shalev-Shwartz, Yoram Singer, and Tushar Chandra. Efficient projections onto the  $l_1$ -ball for learning in high dimensions. In *Proceedings of the 25th international conference on Machine learning*, pages 272–279. ACM, 2008.
- Jonathan Eckstein and Dimitri P Bertsekas. On the douglas-rachford splitting method and the proximal point algorithm for maximal monotone operators. *Mathematical Programming*, 55(1-3):293–318, 1992.
- Shai Fine and Katya Scheinberg. Efficient svm training using low-rank kernel representations. *The Journal of Machine Learning Research*, 2:243–264, 2002.
- James E Gentle. *Matrix algebra: theory, computations, and applications in statistics*. Springer Science & Business Media, 2007.
- Tom Goldstein, Brendan O’Donoghue, Simon Setzer, and Richard Baraniuk. Fast alternating direction optimization methods. *SIAM Journal on Imaging Sciences*, 7(3):1588–1623, 2014.
- Jacek Gondzio and Robert Sarkissian. Parallel interior-point solver for structured linear programs. *Mathematical Programming*, 96(3):561–584, 2003.
- Hans P Graf, Eric Cosatto, Leon Bottou, Igor Dourdanovic, and Vladimir Vapnik. Parallel support vector machines: The cascade svm. In *Advances in neural information processing systems*, pages 521–528, 2004.
- O Güler. Augmented lagrangian algorithms for linear programming. *Journal of optimization theory and applications*, 75(3):445–470, 1992.
- Cho-Jui Hsieh, Si Si, and Inderjit S Dhillon. A divide-and-conquer solver for kernel support vector machines. *arXiv preprint arXiv:1311.0914*, 2013.
- Zakria Hussain and John Shawe-Taylor. Multiple kernel learning: Bounds and algorithms. Personal communication, unpublished manuscript, 2015.
- Thorsten Joachims. Making large scale svm learning practical. Technical report, Universität Dortmund, 1999.
- S Sathiya Keerthi, Olivier Chapelle, and Dennis DeCoste. Building support vector machines with reduced classifier complexity. *The Journal of Machine Learning Research*, 7:1493–1515, 2006.
- Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images, 2009.
- Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

- Hanxi Li and Chunhua Shen. Boosting the minimum margin: Lpboost vs. adaboost. In *Digital Image Computing: Techniques and Applications (DICTA)*, 2008, pages 533–539. IEEE, 2008.
- Chih-Jen Lin. A study on smo-type decomposition methods for support vector machines. *Neural Networks, IEEE Transactions on*, 17(4):893–908, 2006.
- Ji Liu, Stephen J Wright, Christopher Ré, Victor Bittorf, and Srikrishna Sridhar. An asynchronous parallel stochastic coordinate descent algorithm. *arXiv preprint arXiv:1311.1873*, 2013.
- Dhruv Mahajan, S Sathiya Keerthi, and S Sundararajan. A distributed algorithm for training nonlinear kernel machines. *arXiv preprint arXiv:1405.4543*, 2014.
- Jean-Jacques Moreau. Proximité et dualité dans un espace hilbertien. *Bulletin de la Société mathématique de France*, 93:273–299, 1965.
- Dung Duc Nguyen, Kazunori Matsumoto, Yasuhiro Takishima, and Kazuo Hashimoto. Condensed vector machines: learning fast machine for large data. *Neural Networks, IEEE Transactions on*, 21(12):1903–1914, 2010.
- Jorge Nocedal and Stephen Wright. *Numerical optimization*. Springer Science & Business Media, 2006.
- Laura Palagi and Marco Sciandrone. On the convergence of a modified version of svm light algorithm. *Optimization methods and Software*, 20(2-3):317–334, 2005.
- Neal Parikh and Stephen Boyd. Proximal algorithms. *Foundations and Trends in optimization*, 1(3):123–231, 2013.
- John Platt et al. Fast training of support vector machines using sequential minimal optimization. *Advances in kernel methodssupport vector learning*, 3, 1999.
- Danil Prokhorov. Ijcnv 2001 neural network competition. *Slide presentation in IJCNN*, 1, 2001.
- Ali Rahimi and Benjamin Recht. Random features for large-scale kernel machines. In *Advances in neural information processing systems*, pages 1177–1184, 2007.
- Ali Rahimi and Benjamin Recht. Weighted sums of random kitchen sinks: Replacing minimization with randomization in learning. In *Advances in neural information processing systems*, pages 1313–1320, 2009.
- Gunnar Rätsch, Bernhard Schölkopf, A Smola, K-R Müller, Takashi Onoda, and Sebastian Mika.  $\nu$ -arc: Ensemble learning in the presence of outliers. In *ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS 12*. Citeseer, 1999.
- Gunnar Rätsch, Bernhard Schölkopf, Alex J Smola, Sebastian Mika, Takashi Onoda, Klaus-Robert Müller, PJ Bartlett, B Schölkopf, D Schuurmans, et al. Robust ensemble learning. In *Advances in large margin classifiers*, pages 207–220. MIT Press, 2000.
- Robert E Schapire and Yoav Freund. *Boosting: Foundations and algorithms*. MIT press, 2012.

- Bernhard Schölkopf, Sebastian Mika, Chris JC Burges, Philipp Knirsch, Klaus-Robert Müller, Gunnar Rätsch, and Alexander J Smola. Input space versus feature space in kernel-based methods. *Neural Networks, IEEE Transactions on*, 10(5):1000–1017, 1999.
- Shai Shalev-Shwartz, Yoram Singer, Nathan Srebro, and Andrew Cotter. Pegasos: Primal estimated sub-gradient solver for svm. *Mathematical programming*, 127(1):3–30, 2011.
- Ingo Steinwart. Sparseness of support vector machines. *The Journal of Machine Learning Research*, 4:1071–1105, 2003.
- Zhuang Wang, Koby Crammer, and Slobodan Vucetic. Breaking the curse of kernelization: Budgeted stochastic gradient descent for large-scale svm training. *The Journal of Machine Learning Research*, 13(1):3103–3131, 2012.
- Steve Webb, James Caverlee, and Calton Pu. Introducing the webb spam corpus: Using email spam to identify web spam automatically. In *CEAS*, 2006.
- Christopher Williams and Matthias Seeger. Using the nyström method to speed up kernel machines. In *Proceedings of the 14th Annual Conference on Neural Information Processing Systems*, number EPFL-CONF-161322, pages 682–688, 2001.
- Mingrui Wu, Bernhard Schölkopf, and Gökhan Bakir. Building sparse large margin classifiers. In *Proceedings of the 22nd international conference on Machine learning*, pages 996–1003. ACM, 2005.
- 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 *Advances in neural information processing systems*, pages 476–484, 2012.
- Yang You, James Demmel, Kenneth Czechowski, Le Song, and Richard Vuduc. Casvm: Communication-avoiding parallel support vector machines on distributed systems. Technical Report UCB/EECS-2015-9, EECS Department, University of California, Berkeley, Feb 2015a.
- Yang You, Haohuan Fu, Shuaiwen Leon Song, Amanda Randles, Darren Kerbyson, Andres Marquez, Guangwen Yang, and Adolfo Hoisie. Scaling support vector machines on modern hpc platforms. *Journal of Parallel and Distributed Computing*, 76:16–31, 2015b.
- Kai Zhang, Liang Lan, Zhuang Wang, and Fabian Moerchen. Scaling up kernel svm on limited resources: A low-rank linearization approach. In *International Conference on Artificial Intelligence and Statistics*, pages 1425–1434, 2012.
- Ruiliang Zhang and James Kwok. Asynchronous distributed admm for consensus optimization. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pages 1701–1709, 2014.
- Kaihua Zhu, Hao Wang, Hongjie Bai, Jian Li, Zhihuan Qiu, Hang Cui, and Edward Y Chang. Parallelizing support vector machines on distributed computers. In *Advances in Neural Information Processing Systems*, pages 257–264, 2008.
- Zeyuan Allen Zhu, Weizhu Chen, Gang Wang, Chenguang Zhu, and Zheng Chen. P-packsvm: Parallel primal gradient descent kernel svm. In *Data Mining, 2009. ICDM’09. Ninth IEEE International Conference on*, pages 677–686. IEEE, 2009.



# Appendix A

## Further algorithmic details for LPSVM

This section explores some details for LPSVM that were not considered critical for the explanation in chapter 4.

### A.1 KKT system for recovering primal variables

The KKT conditions describe necessary conditions for optimality of constrained objectives. In the case of convex optimisation, including linear programs, they are also sufficient. The conditions are summarised below, and can be found for instance in [Boyd and Vandenberghe, 2004]. Consider a convex optimisation problem in the form:

$$\begin{aligned} \min_x \quad & f_0(x) \\ \text{s.t.} \quad & f_i(x) \leq 0, \quad i = 1, \dots, r \\ & h_i(x) = 0, \quad i = 1, \dots, s \end{aligned} \tag{A.1}$$

for optimisation variable  $x \in \mathbb{R}^m$  where the constraints define a nonempty domain. We define the Lagrangian as  $\mathcal{L}(x, \lambda, \nu) := f_0(x) + \sum_{i=1}^r \lambda_i f_i(x) + \sum_{i=1}^s \nu_i h_i(x)$ . Then  $x^*$  is optimal if and only if:

$$\nabla_x \mathcal{L}(x^*, \lambda, \nu) = 0 \quad (\text{A.2a})$$

$$\lambda_i \geq 0 \quad i = 1, \dots, r \quad (\text{A.2b})$$

$$\lambda_i f_i(x^*) = 0 \quad i = 1, \dots, r \quad (\text{A.2c})$$

$$f_i(x^*) \leq 0, \quad i = 1, \dots, r \quad (\text{A.2d})$$

$$h_i(x^*) = 0, \quad i = 1, \dots, s \quad (\text{A.2e})$$

We can group the conditions as primal feasibility (A.2d), (A.2e), dual feasibility (A.2a), (A.2b) and complementary slackness (A.2c).

### A.1.1 KKT system for LPSVM dual

In this section we derive the KKT conditions for the dual problem (D1). To keep notation light,  $H$  is the matrix for a restricted master problem, and  $u$  the conformable dual variable. First we rewrite the problem in standard form using  $x^\top = (u^\top, \beta)$ ,  $\tilde{H} = \begin{bmatrix} H & -\mathbf{1} \end{bmatrix}$ ,  $c = (0, \dots, 0, 1)^\top$ ,  $d = (1, \dots, 1, 0)^\top$ :

$$\begin{aligned} \min_x \quad & c^\top x \\ \text{s.t.} \quad & \tilde{H}x \leq 0 \\ & -x_i \leq 0 \quad i = 1, \dots, m-1 \\ & x_i \leq D \quad i = 1, \dots, m-1 \\ & d^\top x - 1 = 0 \end{aligned}$$

The lagrangian can be written as

$$\mathcal{L}(x, \lambda, \nu) := c^\top x + \lambda_1 \tilde{H}x - \tilde{\lambda}_2^\top x + \tilde{\lambda}_3^\top (x - D\mathbf{1}) + \nu(d^\top x - 1)$$

for  $\tilde{\lambda}_2^\top = [\lambda_2^\top \ 0]$  and  $\tilde{\lambda}_3^\top = [\lambda_3^\top \ 0]$ . The condition of (A.2a) implies

$$\nabla_x \mathcal{L}(x, \lambda, \nu) = c + \tilde{H}^\top \lambda_1 - \tilde{\lambda}_2 + \tilde{\lambda}_3 + \nu d = 0 \quad (\text{A.3})$$

Relabelling  $a := \lambda_1$ ,  $\xi := \lambda_3$  and  $\rho := \nu$ , and enforcing nonnegativity from (A.2b) we recover the primal constraints. Application of complementary slackness conditions (A.2c) give:

$$\lambda_1^\top \tilde{H}x = 0 \quad \Rightarrow \quad a_j \left( \sum_{i=1}^m u_i y_i h_j(x_i) - \beta \right) = 0 \quad j = 1, \dots, p \quad (\text{A.4})$$

$$\tilde{\lambda}_2^\top x = 0 \quad \Rightarrow \quad \left( \sum_{j=1}^p a_j h_j(x_i) + \xi_i - \rho \right) u_i = 0 \quad i = 1, \dots, m \quad (\text{A.5})$$

$$\tilde{\lambda}_3^\top (x - D1) \quad \Rightarrow \quad \xi_i (u_i - D) = 0 \quad i = 1, \dots, m \quad (\text{A.6})$$

From (A.4), all hypotheses which have an inactive minmax constraint, ie  $Hu < \beta$  have an associated  $a_j = 0$ . We also have from (A.6) that if  $u_i < D$ ,  $\xi_i = 0$ . Thus all margin violations have bounded  $u_i = D$ . Finally, (A.5) tells us that if  $u_i > 0$ ,  $H_i^\top a + \xi_i = \rho$  where  $H_i$  is the  $i^{\text{th}}$  column of  $H$ . This means that all nonzero  $u_i$  correspond to examples either on the margin or in violation of it. This last constraint is particularly helpful in determining the primal variables. We shall call examples with  $u_i \in (0, D)$  *free vectors* in line with support vector terminology (eg. [Bottou and Lin, 2007]), and these have the special property of lying precisely on the margin.

First we may set all trivial values of primal variables according to:

$$(A.4) \quad (Hu - \beta)_j < 0 \quad \Rightarrow \quad a_j = 0$$

$$(A.6) \quad u_i < D \quad \Rightarrow \quad \xi_i = 0$$

Define the matrix  $L$  as the current hypothesis matrix  $H$  with all rows  $\{j : a_j = 0\}$  and columns  $\{i : u_i = 0\}$  removed. From (A.5), we construct the following system:

$$\begin{bmatrix} L^\top & Q & -1_{m_r} \\ -1_{p_r}^\top & -0^\top & 0 \end{bmatrix} \begin{bmatrix} a_r \\ \xi_r \\ \rho \end{bmatrix} = \begin{bmatrix} | \\ 0 \\ | \\ 1 \end{bmatrix} \quad (\text{A.7})$$

where subscripts  $r$  denote the remaining quantities after the removal above, and  $Q$  is a matrix of unit basis vectors corresponding to the remaining  $\xi_i$ . The number of quantities remaining after removal is variable and on occasion this system is underdetermined. One strategy in this case is to add the zero duality gap ( $D \sum_{i=1}^m \xi_i - \rho = -\beta$ ).

A smaller system can usually be constructed which involves only the free vectors. This is solved only for  $a$  and  $\rho$ , and  $\xi$  is then calculated directly from these quantities. For stability, we may still wish to solve the larger system. We can exploit the structure of (A.7),

since  $Q$  is semiorthogonal, and therefore much of the gram matrix will be the identity or permutations of  $L$ . The Cholesky decomposition can be calculated efficiently here and used to solve the system.

### A.1.2 Numerical precision

There are a number of problems with solving this system in practice. The complementary slackness conditions require equality tests, but since all variables involved are calculated from an optimisation problem, there are significant issues with numerical precision. There is a qualitative difference between for example  $u_i = D$  and  $u_i = D - \epsilon$  for some small  $\epsilon < 1e-8$ , and arbitrary cutoffs generate spurious results. In particular, the cutoff for condition (A.4) and (A.6) may differ by a few orders of magnitude and an incorrect decision will often lead to negative values from the system (A.7). The problem will usually be encountered in non-sparse scenarios involving  $O(10^3)$  variables, since we must decrease the tolerance of an optimiser as the size of the problem increases, which becomes infeasible. Unfortunately using the smaller free vector system does not help in this case since the consensus of the full set of variables generates more stable results.

Even reasonably involved heuristics fail in some cases. We omit the heuristics used by LPSVM in this report<sup>1</sup>, but we demonstrate the performance on a typical edge case in table A.1. The numerical precision here for  $\delta = Hu - \beta$  is extremely poor and LPSVM performs as well as could be expected. In this case we are unable to distinguish between zero and non-zero  $a_j$  by the value of  $\delta$  (see rows 9 and 18). For these cases the only path forward is to solve indiscriminately for all  $a_j$ , resulting in an underdetermined system. Usually the underdetermined system results in an infeasible solution, leaving three options are available.

1. Naively use the estimated primal variables anyway. There is no guarantee of convergence and the number of columns required by the solver may be infeasible.
2. Solve the system under a non-negativity constraint. A number of bespoke algorithms exist for this as well as possible QP formulations. However, these are slow and inexact.
3. Re-solve the LP in the primal form. This is also considerably slower.

We use (2) when the systems are small, and (3) for larger ones.

---

<sup>1</sup>see source code for function *solveWeights* on <https://github.com/spoonbill/lpsvm>

	$\delta = Hu - \beta$	LPSVM == 0	Actual == 0	LPSVM $\hat{a}$	Actual $a$
1.	0.00000022599	1	1	0.00466	0.01482
2.	0.00000289191	1	1	0.07854	0.09394
3.	0.00000259911	1	1	0.05590	0.06621
4.	0.00000298780	1	1	0.01698	0.02072
5.	0.00000108975	1	1	0.01458	0.01629
6.	0.00000282622	1	1	0.01575	0.01974
7.	0.00014043125	0	0	0	0
8.	0.00047167483	0	0	0	0
9.	0.00000541877	0	1	0	0.00400
10.	0.00000354834	1	1	0.01567	0.01867
11.	0.00000187077	1	1	0.00674	0.01276
12.	0.00000271778	1	1	0.00528	0.01190
13.	0.00078797743	0	0	0	0
14.	0.00007503684	0	0	0	0
15.	0.00314817342	0	0	0	0
16.	0.00000008830	1	1	-0.00021	0.00130
17.	0.00000046038	1	1	-0.00363	0.00059
18.	0.00000121650	1	0	-0.00862	0
19.	0.00000098980	1	1	-0.00211	0.00368
20.	0.00000076205	1	1	0.00700	0.00514
21.	0.00007150376	0	0	0	0
22.	0.00000063351	1	1	-0.00170	0.00653
23.	0.00000078192	1	1	-0.00928	0.00671
24.	0.00000058497	1	1	0.01654	0.01377
25.	0.00000211713	1	1	0.02353	0.02231
26.	0.00031248725	0	0	0	0
27.	0.00329521728	0	0	0	0
28.	0.00073000961	0	0	0	0
29.	0.00153593082	0	0	0	0
30.	0.00000059573	1	1	0.01361	0.01443
31.	0.00000232111	1	1	-0.00865	0.00749
32.	0.00000097956	1	1	0.01172	0.01258
33.	0.00002873041	0	0	0	0
34.	0.00000060458	1	1	0.01105	0.02640
35.	0.00000129643	1	1	0.02899	0.03318
36.	0.00000048109	1	1	0.03645	0.02075
37.	0.00000007648	1	1	0.04667	0.03966
38.	0.00000119207	1	1	0.01639	0.00903
39.	0.00005638889	0	0	0	0
40.	0.00000431502	1	1	0.03581	0.03601
41.	0.00000526766	1	1	0.07752	0.07902
42.	0.00000133123	1	1	0.04034	0.04703
43.	0.00004187806	0	0	0	0
44.	0.00000124728	1	1	0.05381	0.04027
45.	0.00000046250	1	1	0.10452	0.07652
46.	0.00000107102	1	1	0.18297	0.12083
47.	0.00000190403	1	1	0.11320	0.09773

Table A.1: Difficulties in identifying which  $a_j = 0$ . The value of each row of  $Hu - \beta$  is shown alongside whether it should be classified as 0. Both the LPSVM ‘best guess’ and the ground truth are shown as well as the estimated value of  $a$ . Here 1 indicates *true*. This sample is taken from optimising the IJCNN dataset and the dual linprog precision was  $10^{-10}$ .

### A.1.3 Sparsity of weak learners

The choice of  $D$  naturally affects the sparsity of the weak learner added in each epoch. The KKT conditions tell us how many non-zero  $u_i$  we can expect. From (A.6), there can be at most  $\lfloor 1/D \rfloor$  positive margin violations ( $\xi_i > 0$ ). Also (A.5) implies that only free vectors or margin violations can have positive  $u_i > 0$ . We must obtain the number of free vectors, that is vectors on the margin. Equation (A.5) applied to the free vectors  $i$  yields the relation:

$$H_i^\top a = \rho$$

Let  $p$  be the number of non-zero  $a_j$ . Then the dimensionality of this system of equations is  $p$ , and the hyperplane corresponding to  $a$  is defined by  $p$  points. That is, there will be  $p$  points on the margin, or  $p$  free vectors. Datasets may be constructed such that more points lie on the margin, but if  $x_i$  are drawn from a distribution with continuous density, then:

$$\text{number of non-zero } u_i \leq \lfloor 1/D \rfloor + p$$

Hence, the density of the weak learners is inversely proportional to the value of  $D$ , and will increase by 1 for every iteration. If LPSVM is run for  $t$  epochs, then the maximum number of support vectors is:

$$n_{sv} \lesssim \frac{t}{D} + \frac{1}{2}t^2$$

## A.2 Active set growth by iteration

Included are a few experimental results that support the absence of an optimal growth size  $N$ . Firstly we explore the scaling properties of iterations and the candidate set size,  $|\mathcal{I}'|$  as the size of the dataset increases. Figures A-1 and A-2 show the results of this experiment varying the dataset size from  $m = 5$  and 55 ('000) using MNIST. Results are included when  $H$  has  $p = 3, 10$  and 20 rows, and for  $D = 0.2$  and 0.02 (figures A-1 and A-2 respectively). The rate of change of iterations decreases rapidly, and there is not much advantage using a value  $N > 100$ . Notably the difference in behaviour between  $m = 5$  and 55 ('000) is negligible insofar as a fixed choice of  $N$  for small  $m$  would work equally optimally for larger  $N$ . We do not claim this behaviour holds necessarily for larger  $m$ .

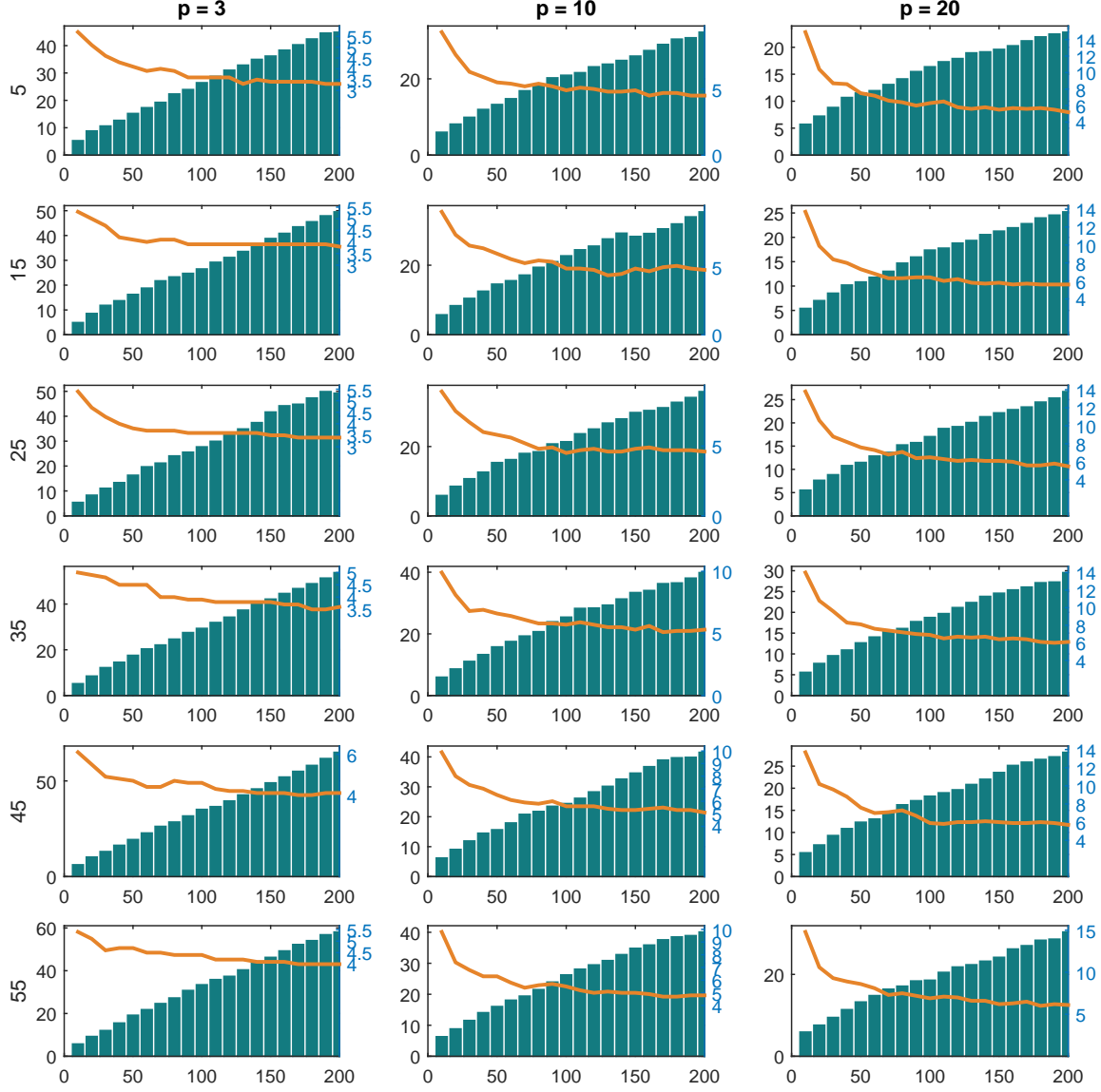


Figure A-1: ( $D = 0.2$ ) The effect of  $N$  upon the candidate set size and number of iterations. Candidate set size as a multiple of the true active set size plotted as bars (LHS), iterations plotted as line (RHS). The dataset size,  $m$  is displayed on the left ('000), the dimension of  $H$  is displayed at the top. Results averaged over 10 trials.

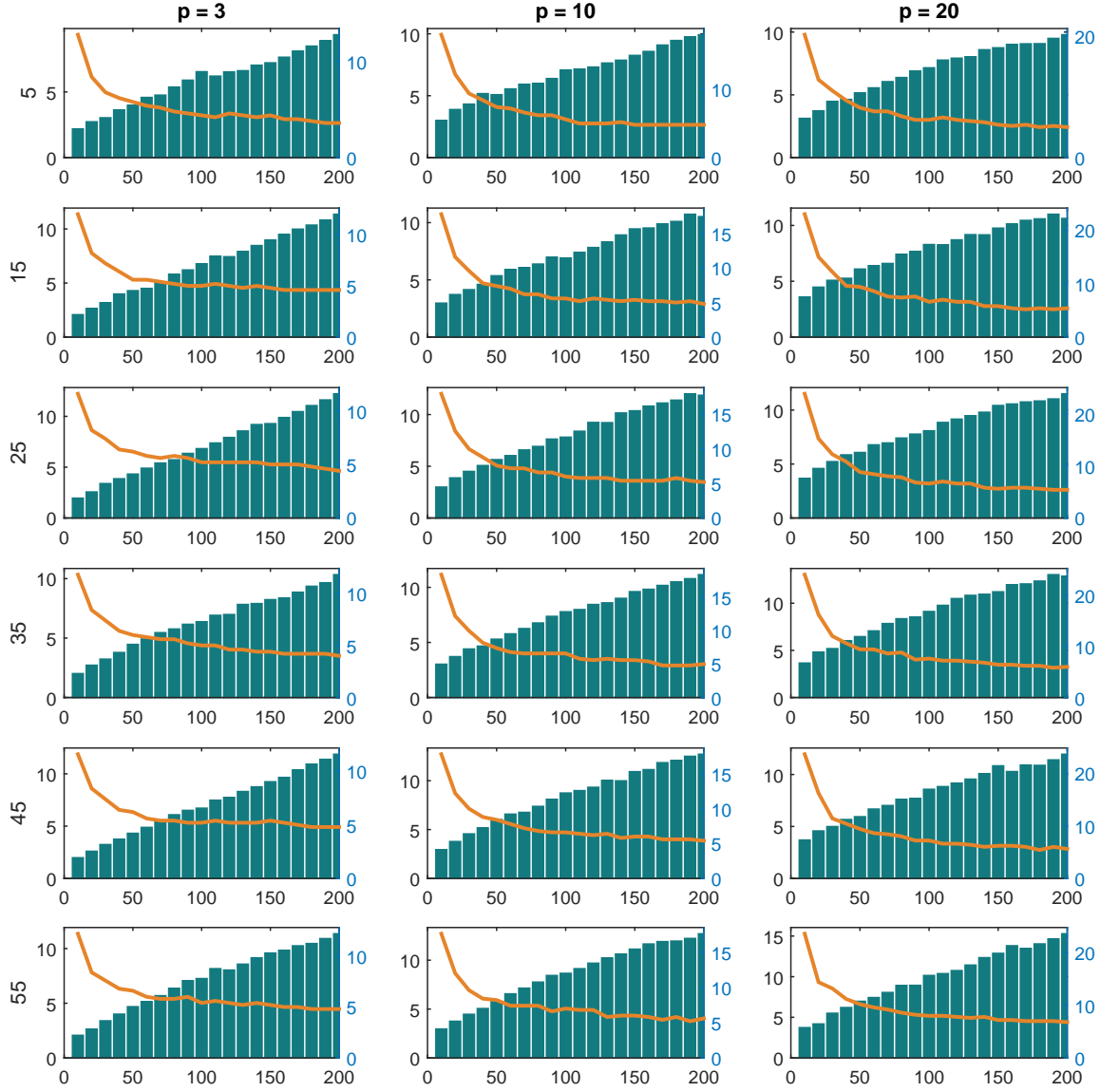


Figure A-2: ( $D = 0.02$ ) The effect of  $N$  upon the candidate set size and number of iterations. Candidate set size as a multiple of the true activeset size plotted as bars (LHS), iterations plotted as line (RHS). The dataset size,  $m$  is displayed on the left ('000), the dimension of  $H$  is displayed at the top. Results averaged over 5 trials.



The idea of a constant  $N$  has been trialled on LPSVM for various values and tested according to wall clock time. Unfortunately for single core experiments the majority of the clock time is spent on kernel computations (over 96% for the  $D = 0.02$  experiments), so results are time consuming and unreliable. For our experiments the time difference in choosing  $N = 20$  vs  $N = 200$  is not statistically significant. It must be emphasised that as  $D$  becomes smaller,  $N$  must necessarily be larger, since the size of the active set grows in general as  $O(1/D)$ . The effect is not noticeable in our experiments, since the size of the active set is only between  $50 - 70$  for  $D = 0.02$  and the size of the candidate set is many multiples of this when the solution is found.

So as in the main text we conclude that our experiments cannot inform the behaviour in anticipation of the distributed algorithm. If the cluster can tolerate the communication of a candidate set size  $5/D$ , then the results may hold for the larger setting. But if we must be more selective, the scaling can be extremely poor, as small values of  $N$  are observed to scale linearly as the column dimension  $p$  increases. In this setting,  $N$  may have clearly defined optimal values since there is a measurable cost associated both with iterations and candidate set size.



## Appendix B

# Details of alternative optimisation methods

In this appendix, we present the methods and results of experiments into alternative optimisation. A summary can be found in chapter 6. We introduce augmented lagrangian methods first. This serves as a sanity check for ADMM, but is also worthy of investigation as the most stable alternative to simplex and interior point methods for constrained linear programming. For convenience, we restate both the primal and dual problems below:

$$\begin{aligned} \min_{a, \rho, \xi} \quad & D \sum_{i=1}^n \xi_i - \rho \\ \text{s.t.} \quad & H^\top a + \xi \geq \rho \\ & a \in \Delta_p, \quad \xi \geq 0 \end{aligned} \tag{P1}$$

$$\begin{aligned} \min_{u, \beta} \quad & \beta \\ \text{s.t.} \quad & H u \leq \beta \\ & 0 \leq u \leq D, \quad \sum_i u_i = 1 \end{aligned} \tag{D1}$$

### B.1 Introduction to penalty and augmented lagrangian methods

The main idea of penalty methods is to harness the power of mature unconstrained optimisation solvers in constrained problems. For a generic optimisation problem,

$$\min_x f(x) \quad \text{subject to } h_i(x) = 0, \quad i \in \mathcal{E} \tag{B.1}$$

where  $\mathcal{E}$  is the index set of equality constraints, consider replacing problem (B.1) with the problem

$$\lim_{c_k \rightarrow \infty} \min_x f(x) + c_k \sum_{i \in \mathcal{E}} |h_i(x)|^2 \quad (\text{B.2})$$

where we have chosen a quadratic penalty. Other positive penalties can be used, such as  $\ell_1$  penalties or Bregman divergences. Under very reasonable technical conditions<sup>1</sup>, we may move the limit inside the minimisation and recover the identical problem.

Nevertheless, this may be of some concern to those worried about efficiency, we have replaced a single optimisation objective with a *sequence* of optimisation objectives to solve. Speed can be retained in part by warm starting each iteration with the solution to the previous, and together with good parameter choices, this can yield a competitive algorithm. The MINOS and LANCELOT solvers use penalty method techniques for nonlinear optimisation. A more serious with these methods is ill conditioning - as  $c_k$  grows large, the Hessian near the minimiser becomes arbitrarily ill-conditioned (§17 in [Nocedal and Wright, 2006]).

A convenient way to deal with the problem of conditioning is to introduce Lagrange multipliers into the problem. Thus (B.2) becomes:

$$\lim_{c_k \rightarrow \infty} \min_x f(x) - \sum_{i \in \mathcal{E}} \lambda_i h_i(x) + c_k \sum_{i \in \mathcal{E}} |h_i(x)|^2 \quad (\text{B.3})$$

The introduction of Lagrange multipliers corrects for systematic bias in the satisfaction of the equality constraints with finite  $c_k$ , and one can show that  $c_k$  need not be increased indefinitely to ensure convergence [Bertsekas, 1996]. Solving the problem (B.3) with a finite sequence  $\{c_k\}$  is known as the *augmented Lagrangian* (AL) method (or *method of multipliers*). It is provably convergent under the same conditions as generic penalty methods, namely that  $f$  and  $h_i$  are continuous, and the minimisation is over a closed set. We can therefore expect this approach to be successful for the LPSVM problems (P1),(D1).

When applying augmented Lagrangian methods to LPSVM, it is easier to work with equality constrained problems, so we reframe (D1) as the following:

$$\begin{aligned} \min_x \quad & f^\top x \\ \text{s.t.} \quad & Ax = b \\ & x \geq 0 \end{aligned} \quad (\text{B.4})$$

where  $A$ ,  $b$ ,  $f$  and  $x$  have the following structure:

---

<sup>1</sup> $f$  and  $h_i$  are continuous over a closed domain [Bertsekas, 1996].

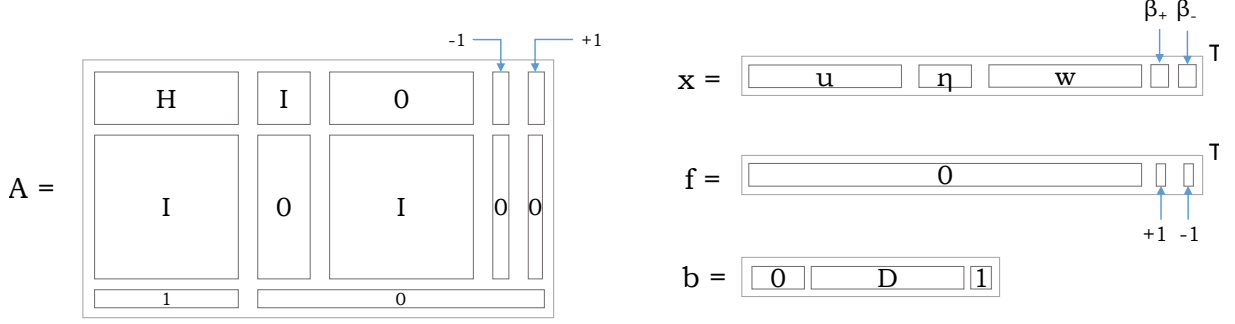


Figure B-1: Structure of the quantities in LP (B.4)

The augmented Lagrangian function for (B.4) is defined as:

$$\mathcal{L}_\tau(x, \lambda) = f^\top x + y^\top (b - Ax) + \frac{\tau}{2} \|Ax - b\|_2^2$$

where  $\lambda$  are the Lagrange multipliers and  $\tau$  the penalty coefficient. Iterating the following scheme will converge to a solution  $x^*$  of the original LP (B.4):

$$x^{k+1} := \operatorname{argmin}_{x \geq 0} \mathcal{L}_{\tau_k}(x, y) \quad (\text{B.5a})$$

$$\lambda^{k+1} := \lambda^k + \tau_k (b - Ax^{k+1}) \quad (\text{B.5b})$$

for a choice of initially small and nondecreasing  $\tau_k^2$ . If we have a good estimate of  $\lambda$ , a good estimate of  $x^*$  can be made even when  $\tau_k$  is not particularly large (Nocedal & Wright Theorems 17.5, 17.6). Thus we have two methods of increasing convergence - increasing the accuracy of  $\lambda$ , and increasing the value of  $\tau_k$ . There are a number of good schemes to exploit these ideas, such as [Güler, 1992], which applies Nesterov's optimal acceleration to the choice of  $\lambda$  and  $\tau$ , or Algorithm 17.4 in [Nocedal and Wright, 2006], both of which performed well in practice.

As expected, the optimisation scheme was shown to work in practice, if slowly. Notably, the objective is now strictly convex (excepting singularity in  $A$ ). In each iteration of (B.5) or an accelerated variant, we must solve the quadratic program associated with  $\mathcal{L}_\tau(x, \lambda)$ . Of course we may simply use a generic QP solver, but this proved to be slow in practice. In the following discussion, we must stress that under our 'Big Data' assumption, we cannot look beyond first order methods for minimising  $\mathcal{L}_\tau(x, \lambda)$ . We therefore look at the gradient

---

<sup>2</sup>see eqn (10) in [Bertsekas, 1996]

with respect to  $x$ :

$$\nabla_x = f - A^\top \lambda + \tau A^\top Ax - \tau A^\top b \quad (\text{B.6})$$

First order optimality may be obtained by solving the above in a linear system. This will however be complicated by the non-negativity constraint on  $x$ , and it is also not an inherently parallelisable operation. Moreover, the system is singular, and even if (D1) has a unique solution, it is only the non-negativity constraint that ensures the problem is well-posed. We choose to consider co-ordinate descent (CD) over gradient descent (GD), since in this framework, CD can operate asynchronously (see for instance [Liu et al., 2013]). Rewriting (B.6) in scalar terms,

$$\mathcal{L}_\tau(x, \lambda) = \sum_{i=1}^n f_i x_i + \sum_{j=1}^p \lambda_j \left( b_j - \sum_{i=1}^n a_i^{(j)} x_i \right) + \frac{\tau}{2} \sum_{j=1}^p \left( b_j - \sum_{i=1}^n a_i^{(j)} x_i \right)^2$$

where  $a_i^{(j)}$  represents the  $i^{\text{th}}$  column of the  $j^{\text{th}}$  row of the matrix  $A$ . Now,

$$\begin{aligned} 0 = \frac{\partial}{\partial x_i} &= f_i - \sum_{j=1}^p \lambda_j a_i^{(j)} + \tau \sum_{j=1}^p a_i^{(j)} \left( b_j - \sum_{i=1}^n a_i^{(j)} x_i \right) \\ &= f_i - A_i^\top \lambda + \tau A_i^\top (b - Ax) \\ &= \underbrace{f_i - A_i^\top \lambda + \tau A_i^\top b}_{\tau m_j} - \tau A_i^\top (A_{\neg i} x_{\neg i} - A_i x_i) \\ \Rightarrow x_i &= \frac{m_i - A_i^\top A_{\neg i} x_{\neg i}}{\|A_i\|^2} = \frac{m_i - A_i^\top Ax}{\|A_i\|^2} + x_i \end{aligned}$$

with  $A_{\neg i}$ ,  $x_{\neg i}$  denoting their respective quantities with the index  $i$  removed. The updates for each  $x_i$  are relatively simple and a simple co-ordinate descent algorithm can be applied with complexity  $O(pn)$  over each batch iteration. If cached, care must be taken to maintain the quantity  $Ax$  under each iteration, since all-at-once co-ordinate descent algorithms are not guaranteed to converge.

This scheme can be refined by utilising the structure of  $A$ . We partition the vector  $x$  into the blocks  $x = (x_u, x_\eta, x_w, x_\beta)$  (see fig B-1), and the corresponding updates are shown below (updates can be derived by basic block linear algebra). Note that the updates for  $\eta$  and  $w$  take on a very simple form, and the  $u$  updates, while more intimidating, comprise largely of constants. All updates are subject to projection steps.

$$\begin{aligned}
(\mathbf{x}_u) \quad x_{u_i} = & \frac{(D+1) + \frac{1}{\lambda}(H_j^\top y_\eta + y_{p+i} + y_{p+n+1})}{\|H_i\|^2 + 2} \\
& - \frac{((H^\top H)_i + \mathbf{1}^\top) x_u + x_{u_i} + H_i^\top x_\eta + x_{w_i} - \beta \sum_j H_j}{\|H_i\|^2 + 2} + x_{u_i}
\end{aligned}$$

$$(\mathbf{x}_{\eta_i}) \quad x_{\eta_i} = b_{\eta_i-n} + \frac{1}{\lambda} y_{\eta_i-n} + \beta - H_{\eta_i-n}^{\text{row}} x_u$$

$$(\mathbf{x}_{w_i}) \quad x_{w_i} = D + \frac{1}{\lambda} y_{p+i} - x_{u_i}$$

$$(\mathbf{x}_{\beta+}) \quad x_{\beta+} = -\frac{1}{\lambda} \overline{y_{1:p}} - \frac{1}{\lambda p} + \overline{H x_u} + \overline{x_\eta} + x_{\beta-}$$

$$(\mathbf{x}_{\beta-}) \quad x_{\beta-} = \frac{1}{\lambda} \overline{y_{1:p}} + \frac{1}{\lambda p} - \overline{H x_u} - \overline{x_\eta} + x_{\beta-}$$

In practice this scheme was still too slow and scaled poorly. The issue was primarily that the optimal step sizes were extremely small and progress extremely slow. This is exacerbated when  $\tau$  is large, since the quadratic term therefore dominates, but the matrix  $A$  will under most circumstances be wide and have a large null space. Thus the gradient information from the linear terms may be very small compared to the ‘valley’-like structure of the quadratic penalty (see fig B-2). Future attempts may abandon strict descent in favour of larger step lengths with quicker convergence. More could be done to understand whether minimising multiple directions at once, or relaxing the simplex constraint on  $u$  would yield better performance. Another avenue of approach may be using Bregman divergences for the penalty, since these have proved efficient in L1 constrained optimisation.

## B.2 Alternating Direction Method of Multipliers

Utilising the fact that the augmented Lagrangian approach is proved to converge for our problem, we now turn to the *alternating direction method of multipliers* (ADMM), which converges under similar conditions. ADMM is a more general framework (which encompasses AL approaches) based on operator splitting, allowing the decomposition of challenging problems into simpler ones. Our primary reason for investigating ADMM is to enable distributed optimisation, which has been apparently unavailable through use of our previous methods. Following the presentation in the excellent review by Boyd et al. [Boyd et al.,

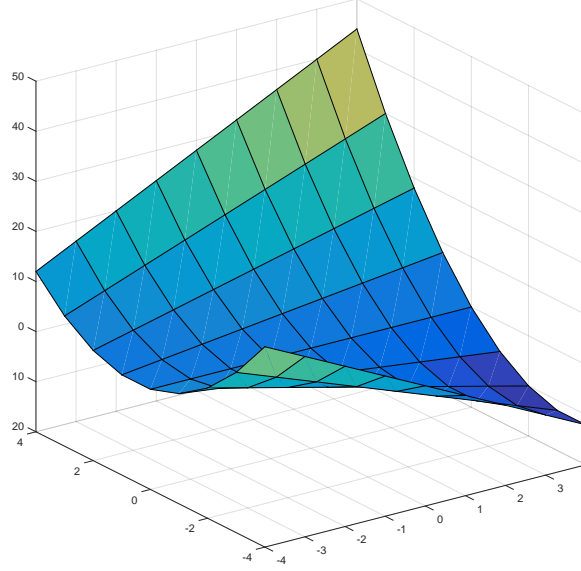


Figure B-2: Example loss surface of a 2D singular QP.

2011], ADMM applies to the following problem:

$$\begin{aligned} \min_{x,z} \quad & f(x) + g(z) \\ \text{s.t.} \quad & Ax + Bz = c \end{aligned} \tag{B.7}$$

The algorithmic scheme involves a minimisation over  $x$  and  $z$  separately, allowing the decomposition of a complex function into easier subproblems. A quadratic regulariser ensures the convergence of  $x$  and  $z$  to the primal constraint  $Ax + Bz = c$ . This simple scheme is surprisingly general. We require only that  $f$  and  $g$  be closed proper convex functions, meaning in particular that nonsmooth or extended real functions can be used. This section uses the framework in the following ways: consensus between  $x$  and  $z$  (ie.  $A = I$ ,  $B = -I$ ,  $c = 0$ ), and splitting the constraint set from the objective function using  $g(z)$ . The ADMM updates are:

$$x^{k+1} := \operatorname{argmin}_x \left( f(x) + \frac{\tau}{2} \|x - z^k + u^k\|_2^2 \right) \tag{B.8a}$$

$$z^{k+1} := \operatorname{argmin}_z \left( g(z) + \frac{\tau}{2} \|x^{k+1} - z + u^k\|_2^2 \right) \tag{B.8b}$$

$$u^{k+1} := u^k + x^{k+1} - z^{k+1} \tag{B.8c}$$

Both the  $x$  and  $z$  updates may be recognised as the proximal operators of  $(1/\tau)f(x)$  and  $(1/\tau)g(z)$ , and due to the choice of constraint,  $u$  is an auxiliary control signal which measures the cumulative non-consensus of  $x$  and  $z$ . The updates minimise their respective functions



near the most recent cumulative primal residual, which results in convergence of  $x$  and  $z$  provided the above conditions are satisfied. For more details refer to [Boyd et al., 2011, Parikh and Boyd, 2013, Eckstein and Bertsekas, 1992].

### B.2.1 ADMM for Linear Programming

We have in [Boyd et al., 2011] that an equality constrained Linear Program,

$$\begin{aligned} \min_x \quad & c^\top x \\ \text{s.t.} \quad & Ax = b, \quad x \geq 0 \end{aligned}$$

can be expressed as:

$$\begin{aligned} \min_{x,z} \quad & f(x) + g(z) \\ \text{s.t.} \quad & x = z \end{aligned} \tag{B.9}$$

$x \in \mathbb{R}^n$ ,  $z \in \mathbb{R}^n$ , where  $f(x) = c^\top x$  with  $\text{dom } f = \{x : Ax = b\}$  ( $+\infty$  outside of its domain) and  $g(z)$  as the indicator function of the non-negative orthant  $\mathbb{R}_+^n$ , and so can be solved with the following updates, as stated in [Boyd et al., 2011]:

$$\text{solve:} \quad \begin{bmatrix} \tau I & A^\top \\ A & 0 \end{bmatrix} \begin{bmatrix} x^{k+1} \\ \nu \end{bmatrix} = \begin{bmatrix} \tau(z^k - u^k) - c \\ b \end{bmatrix} \tag{B.10a}$$

$$z^{k+1} := (x^{k+1} + u^k)_+ \tag{B.10b}$$

$$u^{k+1} := u^k + x^{k+1} - z^{k+1} \tag{B.10c}$$

The system in (B.10a) is obtained easily from the KKT conditions corresponding to (B.8a). By caching a factorisation of the matrix in (B.10a), the scheme (B.10) is a useful linear programming routine which requires minimal coding except familiarity with BLAS routines. If an analytic form of the matrix factorisation is known, further efficiencies may be implemented. For example, implementing this scheme for the active set solutions in LPSVM, the Cholesky factorisation can be shown to be extremely sparse (in an average case over 90% are the elements are block diagonal matrices).

For this project, while much of the Cholesky matrix was analytically derived, the corresponding sparse implementation was not investigated. However, in our MATLAB implementation, an accelerated variant of ADMM based on Nesterov's optimal algorithm derived in [Goldstein et al., 2014] was used. The implementation was nevertheless significantly

slower than MATLAB's `linprog`. However, if convergence can be obtained in a similar number of iterations, but the computation distributed, we might expect a sparse implementation to be competitive.

### B.2.2 Consensus ADMM for Linear Programming

We can obtain a distributed version of the previous algorithm quite simply by making the relevant number of copies of  $x$  and  $z$  for the nodes, and concatenating the vectors. Specifically let  $\mathbf{x} = (x_1^\top, x_2^\top, \dots, x_N^\top)^\top$  and  $\mathbf{z} = (z_1^\top, z_2^\top, \dots, z_N^\top)^\top$  where  $i$  denotes simply the respective copy of the variable.  $x_i$  are permitted to vary across nodes, but we enforce a subset  $s$  of the elements of  $z_i$  to always be in consensus, ie  $(z_i)_s = z_s$  for all  $i$ . We also define  $A_i, b_i$  to be partitions of  $A$  and  $b$  corresponding to the data available in the  $i^{\text{th}}$  partition. Then we pose the consensus problem,

$$\begin{aligned} \min_{\mathbf{x}, \mathbf{z}} \quad & f(\mathbf{x}) + g(\mathbf{z}) \\ \text{s.t.} \quad & \mathbf{x} = \mathbf{z} \end{aligned} \tag{B.11}$$

where  $f(x) = \sum_{i=1}^N f_i(x_i)$ ,  $f_i(x_i) = c_i^\top x_i$  with  $\text{dom } f_i = \{x_i : A_i x_i = b_i\}$  and  $g(\mathbf{z})$  is the non-negative indicator of  $\mathbb{R}_+^{Nn}$ . We apply this framework to (P1) rather than (D1) because the dual cannot be decomposed into a series of functions in consensus. Thus  $A, b, c$  take on different values to those in figure B.4, but the identifiers are dropped to avoid clutter. By enforcing consensus only between each copy of  $(a, \rho)$ , the primal can be shown to admit consensus form. Applying the ADMM updates to problem (B.11) results in a similar scheme to (B.10), but with the linear solve now distributed:

$$\text{solve:} \quad \begin{bmatrix} \tau I & A_i^\top \\ A_i & 0 \end{bmatrix} \begin{bmatrix} x_i^{k+1} \\ \nu \end{bmatrix} = \begin{bmatrix} \tau(z_i^k - u_i^k) - c_i \\ b_i \end{bmatrix} \tag{B.12a}$$

$$(z_i^{k+1})_{\neg s} := \left( (x_i^{k+1})_{\neg s} + (u_i^k)_{\neg s} \right)_+ \tag{B.12b}$$

$$(z^{k+1})_s := \frac{1}{N} \left( \sum_{i=1}^N (x_i^{k+1})_s + (u_i^k)_s \right)_+ \tag{B.12c}$$

$$u_i^{k+1} := u_i^k + x_i^{k+1} - z^{k+1} \tag{B.12d}$$

( $\neg s$  is used in place of the complement operator, ie.  $s^c$  for clarity). In (B.12b) we simply perform a nonnegative projection, but for the consensus variables in (B.12c) there is also an averaging step. The derivation of this follows from the component-wise separability of the LP objective and the quadratic penalty. Note that the average appears in the  $z_s$  update because of the  $\ell_2$  penalty; if we were to use the  $\ell_1$  penalty, the median would be used instead.

The serial ADMM LP efficiency in practice is dominated by repeated triangular back-solve operations<sup>3</sup>. In the distributed version, we replace these by  $N$  smaller systems each of expected size  $1/N$  under uniformly distributed data. Ignoring communication, provided the number of iterations grows at worst linearly with  $N$ , one would expect runtime to decrease linearly as we increase the number of nodes.

### B.2.3 Consensus ADMM by Proximal Evaluation

If we now consider a fully distributed version of (B.11), an analytic form can be derived for each  $x_i$  update. We recast (P1) in hinge loss form:

$$\begin{aligned} \min_{a, \rho} \quad & \frac{1}{n\nu} \sum_{i=1}^n \left( \rho - h_i^\top a \right)_+ - \rho \\ \text{s.t.} \quad & a \in \Delta_p \end{aligned} \tag{B.13}$$

where  $h_i$  is the  $i^{\text{th}}$  column of  $H$ . This is equivalent to

$$\min_{a, \rho} \sum_{i=1}^n \left( \frac{1}{\nu} \left( \rho - h_i^\top a \right)_+ - \rho \right) + g(a)$$

where  $g(a)$  is the indicator function of the  $p$ -dimensional unit simplex. Thus setting  $f_i(a_i, \rho_i) = \frac{1}{\nu} (\rho_i - h_i^\top a_i)_+$ , the problem can be solved by consensus ADMM. The  $x_i$  update is:

$$\left( a_i^{k+1}, \rho_i^{k+1} \right) := \underset{a_i, \rho_i}{\operatorname{argmin}} \left( \frac{1}{\nu} (\rho_i - h_i^\top a_i)_+ + \frac{\rho}{2} \|x_i - z^k + u_i^k\|_2^2 \right)$$

Define  $x_i^\top = (\rho_i^\top, a_i^\top)$ ,  $\tilde{h}_i^\top = (1, -h_i^\top)$ ,  $d_i^k = z^k - u_i^k$ ,  $r = (-1, 0, \dots, 0)^\top$  and set  $\tau = 1$ . Then, recognising the update as a proximal operator, and using Moreau decomposition (see [Moreau, 1965] or [Parikh and Boyd, 2013]),

$$\begin{aligned} x_i^{k+1} &:= \underset{x_i}{\operatorname{argmin}} \left( \frac{1}{\nu} (\tilde{h}_i^\top x_i)_+ + r^\top x_i + \frac{1}{2} \|x_i - d_i^k\|_2^2 \right) \\ &= \mathbf{prox}_{f_i(x_i) + r^\top x_i} \left( d_i^k \right) \\ &= d_i^k - \mathbf{prox}_{(f_i + r^\top)^*(x_i)} \left( d_i^k \right) \end{aligned} \tag{by Moreau}$$

---

<sup>3</sup> $\mathcal{O}(n^2)$ . For typical active set sizes of  $10^3$ , this is true, but since factorisation is  $\mathcal{O}(n^3)$ , this may dominate for larger active sets unless an analytical form is derived

Now we can evaluate  $f_i^*(y)$  as

$$\begin{aligned} f_i^*(y) &= \sup_x \left( y^\top x_i - \frac{1}{\nu} \left( \tilde{h}_i^\top x_i \right)_+ \right) \\ &= \begin{cases} 0 & \text{if } y = \frac{1}{\nu} t \tilde{h}_i, \quad t \in [0, 1]. \\ +\infty & \text{otherwise.} \end{cases} \end{aligned}$$

and therefore  $(f_i + r^\top)^*(y)$  as

$$(f_i + r^\top)^*(y) = \begin{cases} 0 & \text{if } y = \frac{1}{\nu} t \tilde{h}_i - r, \quad t \in [0, 1]. \\ +\infty & \text{otherwise.} \end{cases} =: I_{T_i}(y)$$

where  $T_i$  is the set  $\{(1/\nu)t \tilde{h}_i - r : t \in [0, 1]\}$ . Thus by the definition of **prox** as a generalised projection,

$$x_i^{k+1} := d_i^k - \Pi_{T_i}(d_i^k)$$

We just have a little work remaining to calculate the projection  $\Pi_{T_i}(d_i^k)$  onto the line described by  $T_i$ . The most convenient way to approach the problem is to calculate in parametric form the optimal  $t$  and project onto the interval  $[0, 1]$ . Hence,

$$\begin{aligned} \hat{t} &= \frac{(d_i^k + r)^\top (1/\nu) \tilde{h}_i}{(1/\nu^2) \|\tilde{h}_i\|^2} \\ t &= \frac{1}{\nu} \Pi_{[0,1]}(\hat{t}) \tilde{h}_i \\ x_i^{k+1} &:= d_i^k + r - \frac{1}{\nu} \Pi_{[0,1]} \left( \nu \frac{(d_i^k + r)^\top \tilde{h}_i}{\|\tilde{h}_i\|^2} \right) \tilde{h}_i \end{aligned} \tag{B.14}$$

The full ADMM updates are given below for arbitrary penalty  $\tau$ :

$$x_i^{k+1} := z^k - u_i^k + \frac{1}{\tau} r - \frac{1}{\nu \tau} \Pi_{[0,1]} \left( \nu \frac{(\tau(z^k - u_i^k) + r)^\top \tilde{h}_i}{\|\tilde{h}_i\|^2} \right) \tilde{h}_i \tag{B.15a}$$

$$z_\rho^{k+1} := \frac{1}{n} \sum_{i=1}^n \left( \left( x_i^{k+1} \right)_\rho + \left( u_i^k \right)_\rho \right) \tag{B.15b}$$

$$z_a^{k+1} := \Pi_{\Delta_p} \left( \frac{1}{n} \sum_{i=1}^n \left( \left( x_i^{k+1} \right)_a + \left( u_i^k \right)_a \right) \right) \tag{B.15c}$$

$$u_i^{k+1} := u_i^k + x_i^{k+1} - z_i^{k+1} \tag{B.15d}$$

The  $x$  updates are simple operations consisting of one vector inner product and one vector addition. The simplex projection in (B.15c) can be performed in  $\mathcal{O}(p \log p)$  time, for exam-

ple see [Duchi et al., 2008]. On paper this appears a potentially useful scheme, since the updates are extremely simple, and require sharing knowledge only of  $z$ , a  $(p+1)$ -dimensional vector. We might therefore implement a stochastic consensus ADMM scheme as described in [Zhang and Kwok, 2014]. However such an endeavour is out of scope for this project.

### B.2.4 ADMM experiments

In this section, we examine the results obtained from the distributed linear solve ADMM in section B.2.2. The results of section B.2.1 (ADMM for LPs) have been discarded since there is no advantage in considering such a scheme over the dual-simplex method in the original implementation. The fully distributed proximal method discussed in B.2.3 converges extremely slowly in serial batch form, and could only be made effective by an asynchronous stochastic scheme. The purpose of these experiments is not to obtain fully rigorous results, but to be indicative of the merit of using ADMM for this problem. The experiments can be optimised once this has been established.

**Dataset** As per the goals of LPSVM, we have chosen a task which admits a sparse solution. Experiments are performed over only 1 dataset, MNIST (6 vs Rest); the ‘round vs non-round’ task of the main section is non-sparse. We believe that the results on one dataset are sufficient to gain insight into the characteristics of the performance.

**Competing Algorithms** Parallel implementations of the algorithms have not been implemented, instead the distributed optimisation is run in series, masking the communication overhead. We consider:

1. (LPSVM) Standard *dual-simplex* version of the LPSVM algorithm.
2. (LPSVM-D) Distributed Linear Programming via ADMM.
3. (LPSVM-DLS) Distributed Linear Programming via ADMM with line search.

It is widely recognised that ADMM converges slowly, but it is often the case that ‘reasonable’ solutions can be found after only a few tens of iterations. Unfortunately this is not true in general, and as the number of nodes increases or the dimensions of the problem increase, we have observed that  $O(10^3)$  iterations may be required. Perhaps this is due to the linear/nonsmooth nature of the problem, since for example, the usual quadratic form of SVCs has been observed converging more quickly [Boyd, 2011]. In our experiments we look at the effect of early stopping in order to remove the overhead of parallel computing with large numbers of iterations.

A recurring feature of ADMM convergence is the presence of oscillation in the solution iterates (see fig. B-3). This is in part due to the trade-off in optimising primal feasibility (consensus) and dual feasibility (minimising the distributed objectives). Generally this occurs once the algorithm is near a ‘reasonable’ solution<sup>4</sup>, but this is not always true. A good choice of the parameter  $\tau$  reduces this behaviour, but there is no universally good choice for LPSVM. We were also unable to identify a good strategy for an adaptive choice of  $\tau$ ; suggestions in the literature were not shown to improve performance in general. Oscillations are problematic, since stopping the solver at an arbitrary iteration may result in a solution some way from the optimum.

However, when the solution does oscillate, it usually does so in a 1D subspace in parameter space which passes close to the optimum. In the limit, the subspace contains the optimum, and the oscillation converges, but in looking for early stopping strategies we can still exploit the dimension of this subspace by using line search. This can be done fairly efficiently. In principle, the parameter space is over  $a, \rho$  and  $\xi$ , corresponding to  $\Delta_p \times \mathbb{R} \times \mathbb{R}_+^n$ , but  $\xi$  is a function of  $(a, \rho)$  and in fact, the optimal  $\rho$  is a function of  $a$ , hence we consider the parameter space to be  $\Delta_p$ . The parameters may not be feasible during early stopping, so we project the subspace of  $a$  onto the simplex, and perform a line search over this to find the optimal  $a^*$  (within the subspace). The optimal  $\rho$  can be found by noting that  $\mathbf{1}^\top(\rho - H^\top a^*)_+$  is piecewise linear in  $\rho$  and ordering the values of  $H^\top a^*$  to find the optimal  $\rho^*$ . This strategy is employed in the algorithm (LPSVM-DLS).

**Setup** We run each algorithm over the datasets for 15 boosting iterations (epochs). Both the accuracy (0-1 loss) and time per core are reported in table B.1. We report the solver time, as well as the total time to understand the efficiency as well as the sparsity of the solver. We repeat experiments 5 times and report a 95% confidence interval due to randomness in the initialisation, and variance from the early stopping. All experiments run on a PC with Intel i5 3210M 2.50GHz processor and 8GB RAM.

**Results** The solver time of ADMM for 16 partitions over 250 iterations is still worse than than standard `linprog` solver on a single core. The results also suggest that convergence is much slower using 16 partitions than with 4 partitions, so the speed may not be increased by adding further cores. LPSVM-DLS perversely performs worse on the test set than LPSVM-D. The dual variables must be calculated from the primal via the KKT system, but this is not really valid except at the optimum. Both using the line search strategy or running larger numbers of iterations can result in worse dual solutions, despite a

---

<sup>4</sup>we define a ‘reasonable’ solution as one which motivates a similar addition of constraints in the active set as compared to the optimal solution

	LPSVM
Total time (s)	$37.65 \pm 0.71$
Solver time (s)	$2.47 \pm 0.02$
Test error (%)	$1.52 \pm 0.34$

	LPSVM-D					
Partitions	4			16		
Iterations	250	500	1000	250	500	1000
Total time (s)	$15.27 \pm 1.31$	$22.40 \pm 1.07$	$32.43 \pm 1.57$	$5.61 \pm 0.09$	$9.84 \pm 0.42$	$17.64 \pm 1.06$
Solver time (s)	$6.87 \pm 0.84$	$15.70 \pm 4.46$	$25.88 \pm 5.96$	$3.48 \pm 0.10$	$8.19 \pm 1.24$	$15.97 \pm 1.56$
Test error (%)	$1.78 \pm 1.02$	$1.99 \pm 0.68$	$1.34 \pm 0.33$	$1.72 \pm 0.39$	$1.80 \pm 0.67$	$1.77 \pm 0.37$

	LPSVM-DLS					
Partitions	4			16		
Iterations	250	500	1000	250	500	1000
Total time (s)	$12.08 \pm 0.65$	$18.68 \pm 0.40$	$32.13 \pm 2.45$	$4.99 \pm 0.12$	$8.99 \pm 0.32$	$16.75 \pm 1.39$
Solver time (s)	$5.82 \pm 0.41$	$12.09 \pm 0.61$	$24.15 \pm 2.14$	$3.39 \pm 0.10$	$6.99 \pm 0.34$	$14.10 \pm 1.17$
Test error (%)	$1.87 \pm 0.78$	$2.29 \pm 0.66$	$1.80 \pm 0.31$	$2.26 \pm 0.57$	$2.22 \pm 0.79$	$2.02 \pm 0.52$

Table B.1: Training time per partition (seconds) and % accuracy under 0-1 loss for MNIST 6 vs Rest.

better primal solution. Thus the decision of when to stop the ADMM solver has somewhat arbitrary results - leading us to recommend optimising the LP to fairly high precision.

We hold these results to be indicative of the behaviour of these algorithms. There are a number of caveats though. Firstly, the batch serial procedure and algorithmic implementation is a large collection of loops in MATLAB, which is clearly suboptimal. Observe that there is less than a  $2\times$  speed-up per core from 4 partitions to 16 partitions. The back-solve operations (accessing `dTRSV` BLAS routines more directly via `linsolve`) should be the bottleneck, but they account for less than 25% of the runtime. Thus we may expect an improvement of  $3 - 4\times$  for a good implementation. On the other hand the parallel computing overheads and communication inefficiency are also ignored. On balance we believe the experiments have shown insufficient evidence that approximate distributed solutions can yield efficiency in a controllable way. Exploring parallel versions of interior point LP solvers may prove a better way to parallelise the operations here. Removing oscillations will help the stability of the ADMM procedure, for which an  $\ell_1$  or Bregman penalty may be more appropriate. Finally investigating adaptive choices of  $\tau$  may also help for both stability and convergence.

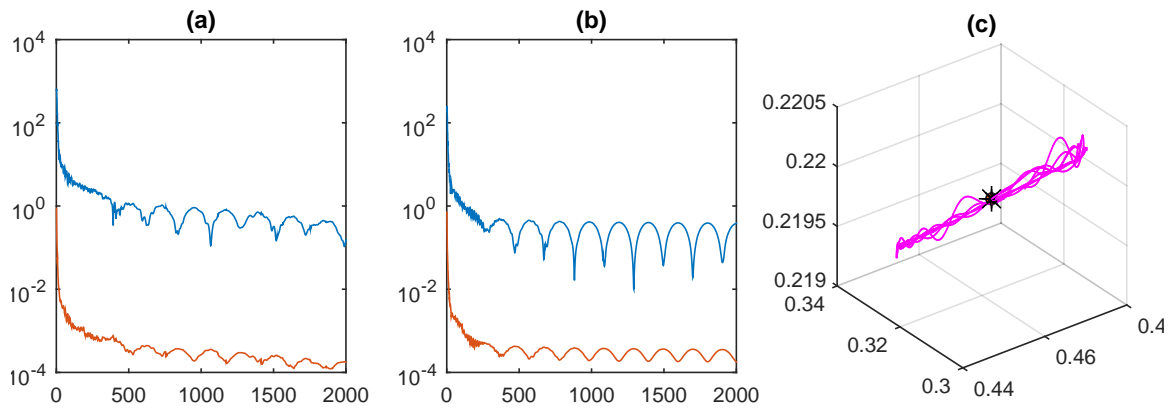


Figure B-3: (a), (b) examples of oscillation in convergence. Red = primal residual, Blue = dual residual. (c) subspace described by  $a \in \Delta_3$  during the oscillations of (b). Optimum in black.