

DEVELOPMENT OF LAGRANGE MULTIPLIER ALGORITHMS FOR TRAINING
SUPPORT VECTOR MACHINES


by

Mayowa K Aregbesola
A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
In Partial fulfillment of
The Requirements for the Degree
of
Doctor of Philosophy
Computational Mathematics

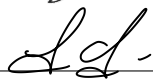
Committee:

_____ 

Dr. Igor Griva, Dissertation Director

_____ 

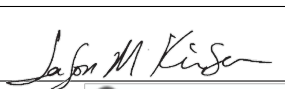
Dr. Jason Kinser, Committee Member

_____ 

Dr. Juan Cebal, Committee Member

_____ 

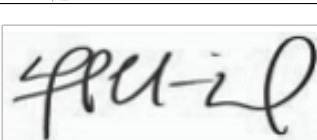
Dr. Maria Emelianenko, Committee Member

_____ 

Dr. Jason Kinser, Department Chairperson

_____ 

Dr. Donna M. Fox, Associate Dean,
Office of Student, College of Science,
Affairs & Special Programs

_____ 

Dr. Fernando R. Miralles-Wilhelm, Dean,
College of Science

Date: _____

Summer Semester 2022
George Mason University
Fairfax, VA

Development of Lagrange Multiplier Algorithms for Training
Support Vector Machines

A dissertation submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy at George Mason University

By

Mayowa K Aregbesola
Master of Science
George Mason University, 2010
Master of Telecommunications
King Fahd University of Petroleum and Minerals, 2005
Bachelor of Science
Obafemi Awolowo University, 2001

Director: Igor Griva, Professor
Department of Computational and Data Sciences

Summer Semester 2022
George Mason University
Fairfax, VA

Copyright © 2022 by Mayowa K Aregbesola
All Rights Reserved

Dedication

I dedicate this dissertation to my dad, Prof. Y.A.S. Aregbesola, the first mathematician to instill the love of numbers in me.

Acknowledgments

Foremost, I would like to thank my advisor Dr. Igor Griva for all of the time he put into mentoring, helping and guiding me through the entire dissertation process. I would like to thank my committee members Dr. Juan Cebal, Dr. Jason Kinser and Dr. Maria Emelianenko, for all of their valuable comments and helpful suggestions.

I would like to thank my wife Olaide, whose unrelenting support and encouragement really made this dissertation possible. I would like to thank my family starting with my parents, Yaya and Odunola, my sisters Opeyemi, Folasade and Motolani for all of their love and support. I would like to thank Dr Raimi Rufai, Ismail Musa and Ganiyu Otukogbe for their encouragements, continuous advice and support.

Table of Contents

	Page
List of Tables	vi
List of Figures	vii
Abstract	ix
1 Introduction	1
2 Support Vector Machine	10
3 SVM Quadratic Programming problem	24
4 Working set selection method for SVM	36
5 Implementation of Lagrange multiplier SVM solvers	47
6 Results	66
7 Discussions	110
A Appendix	115
A.1 SVM and Other classifiers	115
A.2 Source Code	115
Bibliography	116

List of Tables

Table	Page
4.1 WSS combination used in analysis	46
6.1 Data used for testing (Tests 1 - 17).	68
6.2 Data used for testing (Tests 18 - 30).	69
6.3 Training results without decomposition	75
6.4 Comparison of results with decomposition WSS_4	75
6.5 WSS timing versus ALFPGM timing for test 21 (100 pairs)	80
6.6 WSS timing versus ALFPGM timing for Test 28 (100 pairs)	82
6.7 ALFPGM results for Test 16 with varying nPairs	85
6.8 ALFPGM results for Test 25 with varying nPairs	87
6.9 ALFPGM results for Test 30 with varying nPairs	88
6.10 Comparing different FISTA methods for WSS_8 (100 Pairs)	93
6.11 NRAL-Newton results for Test 21 (100 pairs)	94
6.12 NRAL-Newton results for Test 28 (100 pairs)	94
6.13 NRAL-Newton results for Test 16 with varying nPairs	97
6.14 NRAL-Newton results for Test 25 with varying nPairs	99
6.15 NRAL-Newton results for Test 30 with varying nPairs	100
6.16 ALFPGM, NRAL, LibSVM (scikit-learn) comparison - WSS_7	103
6.17 Variation in data set size m and training time ratio with LibSVM	107
6.18 Classification and testing error ijcnn1 using ALFPGM, NRAL and LibSVM	107
6.19 Classification and testing error w8a using ALFPGM, NRAL and LibSVM	108
6.20 NRAL (40 Pairs) vs LibSVM for very large dataset	108
A.1 scikit-learn run for tests w8a and w3a	115

List of Figures

Figure	Page
2.1 Hyperplanes separating data [1]	11
2.2 Maximizing the margin [1]	12
2.3 Linear nonseparable case [1]	18
5.1 Comparison between WSS time and ALFPGM time for webdata_wXa WSS_0	48
5.2 Comparison between WSS time and ALFPGM time for webdata_wXa WSS_4	49
6.1 Comparison between ALFPGM and IPM without decomposition. (MATLAB)	70
6.2 Comparison between IPM and NRAL without decomposition(MATLAB). .	71
6.3 Comparison between ALFPGM and NRAL without decomposition (MATLAB).	71
6.4 Training time comparison between ALFPGM, NRAL, and IPM without decomposition (MATLAB).	73
6.5 Comparison of non-decomposition classification errors (MATLAB).	73
6.6 Comparison of normalized training time ALFPGM, NRAL, and IPM without decomposition (MATLAB).	74
6.7 Comparison between ALFPGM and IPM with decomposition WSS_4 (MATLAB).	76
6.8 Comparison between IPM and NRAL with decomposition WSS_4 (MATLAB).	76
6.9 Comparison between ALFPGM and NRAL with decomposition WSS_4 (MATLAB).	77
6.10 Comparison between training times with decomposition WSS_4 (MATLAB).	77
6.11 Comparison between classification errors (WSS_4).	78
6.12 Normalized Training Time with Decomposition	78
6.13 Comparison between WSS time and ALFPGM time for webdata_wXa WSS_0	80
6.14 Comparison between WSS time and ALFPGM time for webdata_wXa WSS_4	81
6.15 Comparison between WSS types - timings	81

6.16 Comparison between WSS types - Total FPGM Iterations	82
6.17 Performance improvements with using Multiprocessing Test 24 (18201 data points 4295 features)	83
6.18 ALFPGM - Classification time for different number of pairs p - Test 16. . .	84
6.19 ALFPGM - Classification time for different numbers of pairs p Test 25. . .	86
6.20 ALFPGM - Classification time for different numbers of pairs p - Test 30. .	86
6.21 Training time comparison between WSS types and nPairs p	89
6.22 Comparison of the total number of decompositions between WSS types and nPairs p	90
6.23 Comparison between Training times Fista types and the number of pairs - WSS_7 Time	90
6.24 Comparison between Fista types - Classification errors - Time (100 Pairs) .	91
6.25 Comparison between Fista types - Number of decompositions - Time (100 Pairs)	92
6.26 Performance improvements with using Multiprocessing Test 24 (18201 data points 4295 features)	95
6.27 NRAL - Classification time for different pairs - Test 16.	96
6.28 NRAL - Classification time for different pairs - Test 25.	98
6.29 NRAL - Classification time for different pairs - Test 30.	98
6.30 NRAL - Number of decompositions for different number of pairs.	101
6.31 NRAL - Classification time for different working set selection schemes. . .	101
6.32 ALFPGM, NRAL and LibSVM (scikit-learn) timing comparison - WSS_7 .	104
6.33 ALFPGM, NRAL, and LibSVM Classification Error Comparison - WSS_7 .	104
6.34 Comparison of NRAL and ALFPGM results - WSS_7	105
6.35 Comparison of the results of NRAL and LibSVM - WSS_7	105
6.36 Comparison of the results of ALFPGM and LibSVM - WSS_7	106
6.37 Comparison of normalized SVM training times - WSS_7	106

Abstract

DEVELOPMENT OF LAGRANGE MULTIPLIER ALGORITHMS FOR TRAINING SUPPORT VECTOR MACHINES

Mayowa K Aregbesola, PhD

George Mason University, 2022

Dissertation Director: Igor Griva

The Support Vector Machine (SVM) is a supervised learning method that is widely used for data classification and regression. SVM training times can be significant for large training dataset. In pursuit of developing efficient optimization techniques for training SVM with very large datasets, the decomposition method is often used, where the SVM problem is broken into a series of SVM sub-problems. The subset of elements used in each decomposition step, called a working set, needs to be selected efficiently while working toward the goal of finding the full SVM solution.

SVM training time can be further reduced by using parallel processing, allowing the training algorithms to run faster and more reliably. In this work, we used the Augmented Lagrangian Fast Projected Gradient Method (ALFPGM) and the Nonlinear Rescaling Augmented Lagrangian (NRAL) are used for training the SVM subproblems. We developed and implemented parallel algorithms for training SVM and we used optimized matrix-vector, matrix-matrix operations, and memory management to speed up the ALFPGM and NRAL algorithms.

We proposed new working set selection (WSS) schemes to select the working sets used in the SVM decomposition. The results obtained using the proposed WSS show faster training times, while achieving a similar classification error compared to other approaches found in the literature. Numerical results showing SVM training times and classification errors obtained using the ALFPGM and NRAL methods are compared with the results obtained using LibSVM (**sklearn.svm.SVC**), a widely used SVM solver. Numerical results show that faster training times were achieved using NRAL over LibSVM for large dataset SVM problems while achieving similar and in some cases smaller training data classification errors.

Chapter 1: Introduction

Machine learning is the discipline of learning *models* from data using computers. Machine learning problems are divided into two broad categories; supervised and unsupervised learning problems. Learning is called *supervised* when the training data are labeled examples, otherwise it is called *unsupervised*. In supervised learning, a mapping function that maps an input to an output based on known input-output pairs, called training set, is learned. Supervised learning problems can be grouped into regression and classification problems. In classification, the output variable is a category or class, such as a binary classification. Whereas in a regression problem, the output variable is a real value or continuous output. Unlike supervised learning, unsupervised learning uses unlabeled data. After training a classifier from labeled data, a machine can decide on the class of new data it has never seen before with high accuracy. Models learned by supervised learning fall largely into two broad categories: *classifiers* and *predictors*.

Support Vector Machines (SVM) is a widely used supervised learning technique that is used to build classifiers and predictors. Support vector machines (SVMs) are among the most established machine learning algorithms. SVMs were initially developed for binary classification where the output of the learned function is either positive or negative [2–4] and have been extended and used for regression [5] and Rank learning [6]. SVMs works excellently well when there is a clear margin of separation between the different classes. Additionally, SVMs are effective in high-dimensional spaces and in cases where the number of dimensions is greater than the number of samples. SVMs can capture complex, nonlinear decision boundaries with good generalization to previously unseen data. In other words, they have low generalization errors. SVM Multiclass classification can be achieved by combining multiple binary classifiers using the pairwise coupling method [7, 8].

Two special properties of SVMs are that they achieve high generalization by maximizing the margin, and they support efficient learning of nonlinear functions using the kernel trick [1]. One of the main characteristics of SVMs is that the problem is a convex quadratic programming (QP) problem. Thus, the global minima can be found and the problem is readily solvable using quadratic programming techniques. Secondly, the resulting data classifier can be specified completely in terms of its support vectors and kernel function type. Details on the mathematical formulation of SVM are provided in Chapter 2.

SVM Solvers

The Lagrangian dual of the SVM classification problem is a convex quadratic optimization problem. The SVM solution is the optimum of a well-defined convex optimization problem [1, 9]. SVM training requires solving a large-scale convex quadratic optimization problem that has a dense Hessian of the objective function. Due to the convexity of the SVM training problem, finding a global solution is guaranteed as long as the Karush-Kuhn-Tucker (KKT) conditions are satisfied. Therefore, computational efficiency and memory requirements are generally the two main factors to consider when choosing a particular optimization algorithm [9]. General-purpose high-performance optimization packages (MINOS, LOQO, and MATLAB) used to solve quadratic programming problems were initially used to find a solution to SVM problems. However, as Bottou et al. [9] pointed out, there are differences, however, between the SVM problem and the usual quadratic programming benchmarks.

- Quadratic optimization packages were often designed to take advantage of the sparsity in the quadratic part of the objective function. Unfortunately, the SVM kernel matrix is rarely sparse. Sparsity occurs, however, in the solution of the SVM problem, as the number of Support Vectors(SVs) is often much smaller than the number of training samples.
- The specification of an SVM problem rarely fits in memory. The kernel matrix coefficient must be cached or computed on the fly. Vast speed-ups are achieved by accessing

the kernel matrix coefficients with care.

- Generic optimization packages sometimes do extra work to locate the optimum with high accuracy. The accuracy requirements for a learning problem are unusually low.

Since the 1990s, there have been many attempts to develop efficient algorithms for SVM training. First, interior point methods (IPM) [10] have been used. Interior point methods (IPMs) that require solving linear systems of equations such as [10, 11]. Later, the nonlinear rescaling principle has lead to exterior points methods introduced by R. Polyak [12] has been shown to be competitive with IPM [13, 14].

The interior and exterior point methods are based on solving the m -dimensional linear systems of equations, where m is the number of training examples. Therefore, they are efficient for training SVM only up to a few thousand data points. The design of many initial SVM solvers assumes that the full kernel matrix is readily available. Calculating the complete kernel matrix is expensive and unnecessary, so the decomposition methods [15–17] were designed to overcome this difficulty. In pursuit of training SVM on larger data sets, decomposition methods such as sequential minimal optimization (SMO) [18] gained popularity due to their low memory usage requirement and efficiency. Decomposition methods address the full-scale dual problem by solving a sequence of smaller quadratic programming sub-problems. Instead of updating all variables α in the dual space, each iteration of the decomposition method optimizes a subset of $\alpha_i, i \in \beta$, and leaves the remaining coefficients $\alpha_j, j \notin \beta$ unchanged. Most SVM solvers in recent years have been using decomposition techniques to speed up SVM.

There are some challenges that arise when using decomposition.

- Selecting the optimum SVM sub-problem for each iteration to ensure a fast training time.
- Solving the sub-problem optimally.
- Updating the SVM problem gradient with the newly computed variables α .

Sequential Minimal Optimization was proposed by Platt[19], where a working set size of 2 is iteratively selected and the target function is optimized with respect to them. The SMO is one of the most widely used SVM solvers. It has good convergence properties and is easily implemented. The key point is that for a working set of 2 the optimization subproblem can be solved analytically without explicitly invoking a quadratic optimizer. The biggest advantage of this method is that the derivation and implementation are astoundingly simple. However, the drawback is that pairs of training data optimized in this way must be iterated many times. Details on the SMO can be found in the following works [19–22]. In each iteration, the SMO heuristically selects two variables (α_i, α_j) and solves the SVM subproblem using a closed-form solution. The Sequential Minimal Optimization algorithm selects the working set using the maximum violating pair scheme [19] and for each working set, the solution is found. Much of the entire SMO algorithm is dedicated to heuristics to choose which pairs α_i and α_j to optimize to minimize the objective function as much as possible. For large data sets, this is critical to the speed of the algorithm, as there are $m(m - 1)/2$ possible choices for α_i and α_j , and some will result in much less improvement than others.

LibSVM ([22–24]), a very popular tool for training SVM, implements a variant of this method. In general, any small number of variables may be optimized at once, with working set size representing a trade-off between work per iteration and the number of iterations required. LibSVM utilizes the steepest feasible descent approach for selection of the working set and introduces kernel caching and shrinking strategies to accelerate the speed of SVM training. However, it costs a lot of time to select the working set for decomposition and SMO. In each of the optimization iterations, the two approaches should update the entire training set and select the current working set, which contributes to minimize the objective function in the current step.

Working set selection using 2^{nd} order statistics [23] has been used for working set selection and is now the most widely used working set schemes. There have been several works [25–27] that have attempted to further speed up SMO algorithms by having them run in a

parallel architecture.

Most parallel SVM implementations for multicore or GPU systems use the explicit parallelization approach on 2-pair decomposition methods [22, 28–31]. In Sha et al. [32] introduced a multiplicative update rule for SVM optimization problem that uses large matrix-vector multiplications in each iteration, and Chapelle [33], proposed a primal formulation for the least squares hinge loss which results in matrix-matrix and matrix-vector operations. Both works ([32, 33]) are cases of implicit parallelization by reducing the kernel SVM optimization to dense linear algebra operations.

There are several parallel implementations of SVM solvers based on two-pair decomposition aimed at multicores. Some methods attempt to extract the existing parallelism from the SMO-based approaches [34, 35], including a simple modification of LibSVM that computes kernel matrix entries in parallel with OpenMP. Other approaches attempt to do some restructuring of the problem. Increasing the size of the working set (originally two variables in SMO) exposes additional parallelism, as several dual variables are optimized at each iteration [36–38], as well as optimizing nested working sets [39]. Another common approach is to partition the training set, optimize the partitions in parallel, and combine the resulting solutions [26, 40–43].

Likewise, all previous attempts to accelerate the training of kernelized SVM on GPUs have been direct implementations of a 2-pair decomposition method such as SMO. GPU SVM offloads the computation of kernel matrix rows to the GPU using the CUBLAS library and computes KKT condition updates on the GPU with explicitly parallelized routines [30]. Carpenter et al. [29] demonstrated a similar approach and demonstrated the results. In recent times, however, some further attempts to improve the SMO have led to the appearance of decomposition methods that use larger working sets than does the SMO (e.g. GTSVM [31] and the IPSMO-2 algorithm in [27]). GTSVM takes the strategy of increasing the working set size of dual variables to 16 to better utilize GPU resources. However, they focused only on datasets whose kernel matrices can fit into computer memory. IPSMO-2

allows a selection of a larger size of a working set to fit comfortably in an operating memory of a node of a high-performance computing (HPC) multi-node system.

On the other hand, in the last decade, fast gradient methods related to the family of optimal first-order methods introduced by Nesterov [44], have gained their popularity. Beck and Teboulle analyzed the convergence of the FISTA algorithm [45, 46]. Later, R. Polyak established convergence bounds for fast projected gradient methods (FPGM) [47, 48]. In particular, he analyzed the FPGM convergence for solving a nonnegative least squares problem. The theory developed in [48] can be used to justify the convergence of the FPGM for the minimization of a general convex quadratic function for bounded variables.

Training SVM requires solving a large-scale convex quadratic problem with a linear constraint and simple bounds for variables. Therefore, the fast projected gradient method is a natural choice for solving the SVM problem, since projection on the set defined by simple bounds is computationally inexpensive. The linear constraint can be enforced with the help of the augmented Lagrangian method. The theoretical analysis of the corresponding augmented Lagrangian Fast Projected Gradient Method can be found in [49]. Similarly, Polyak et al. have studied Nonlinear Rescaling with SVM [50]. We hope to extend these works by applying decomposition techniques and studying how the algorithm performs in relation to SMO.

Good tutorial introductions to SVM training methods can be found in [1, 5, 8, 9, 22, 51].

Contributions

In this work, we have considered a new approach to training SVM different from the 2-pair decomposition approach by using the decomposition technique with a working set size greater than 2. The hypothesis is that by increasing the number of pairs to more than 2, we can, by using efficient quadratic solvers, reduce the overall training time of SVM. Numerical results have demonstrated faster training times with working set size greater than 2. To solve the decomposition problem, we have used the Augmented Lagrangian Fast projected

gradient method (ALFPGM) and Nonlinear Rescaling Augmented Lagrangian (NRAL) as the quadratic problem solver. The goal is to utilize improved computing capabilities and cheaper memory storage to reduce SVM training times for large training dataset SVM problems.

Our contributions in this work come in four parts.

- We developed an improved ALFPGM for training SVM using decomposition techniques. We have developed and implemented ALFPGM with high-performance computing techniques to reduce training time. We compared different FISTA algorithms in ALFPGM for SVM training. To the best of our knowledge, no one has used ALFPGM for SVM training using decomposition methods. We have seen works like Han et al. [52] and Bloom et al. [14] that used FISTA based algorithms for SVM training for small data set problems whose kernel matrices fit into the computer memory.
- We developed Nonlinear Rescaling Augmented Lagrangian (NRAL) with newton for the SVM decomposition solution. We have developed and implemented NRAL with high performance computing techniques to reduce training time. Previous works like Polyak et al. [50] have studied Nonlinear Rescaling with SVM. The difference with our work, is that we have applied the decomposition techniques and have expanded the area by studying how the algorithm performs with different decomposition methods and working set sizes.
- We proposed and developed working set selection schemes ($WSS_3, WSS_4, WSS_7, WSS_8$) to select the p pairs elements of the working set where $p > 2$ to reduce SVM training times. Previously, Maximum Violating Pair (MVP) and working set selection using second-order statistics have been used for working set selection. Some works such as [27] have used pairs $p > 2$ in the selection of the working set. In this thesis, we present four new working set selection schemes.
- Finally, we studied how different parameter selections affect the SVM training time

and classification errors. We examined how the choice of p pairs affects the overall training results and gave recommendations on choosing the optimal parameters.

Contents

This thesis is structured as follows. Chapter 2 reviews the mathematical formulation of SVM. Chapter 3 presents the decomposition technique and the selection of working sets. Chapter 4 discusses quadratic optimization problems and different FISTA algorithms. Chapter 5 discusses the Augmented Lagrangian Fast Projected Method (ALFPGM) and Nonlinear Rescaling Augmented Lagrangian (NRAL) for SVM problems. Chapter 6 presents numerical results, and finally chapter 7 discusses our conclusions.

Notation

To facilitate the reading of this thesis, here are some important notations.

\mathbf{D} : dataset

\mathbf{x}_i : n -dimensional real vector

y_i the binary class to which the point \mathbf{x}_i belongs (1 or -1)

m : size of training samples

n : dimension of the samples feature

nD : Number of decomposition steps

nSv : Number of support vectors

$nBSv$: Number of border support vectors

\mathbf{w} : weights vector

$\boldsymbol{\alpha}$: Lagrange multipliers vector

b : bias term

Q : kernel matrix

B : working set

\mathbf{e} : the vector of all ones

C : regularization parameter

Chapter 2: Support Vector Machine

A Support Vector Machines (SVM) is a supervised classification technique that is used for learning classification and regression. SVMs were originally developed for classification [4] and have been used for regression [5]. The original form of SVM is a binary classifier in which the output of the learned function is either positive or negative. Multiclass classification can be implemented by combining multiple binary classifiers using the pairwise coupling method [7, 8]. Two special properties of SVM are that they achieve high generalization by maximizing the margin and that they support efficient learning of nonlinear functions using the kernel trick [1]. A.1 shows a comparison between the run-time and accuracy of different machine learning classifiers in the **scikit-learn** machine learning tool. SVM in the examples presented have lower training and testing data misclassification errors.

Hard-Margin SVM Classification

In a binary classification scenario, a classifier seeks to separate training data into two categories. SVMs perform classification by finding a hyperplane that separates or differentiates the two classes. The training set is said to be linearly separable when there exists a linear hyperplane that separates the classes of all training examples. When a training set is linearly separable, there are an infinite number of separating hyperplanes. To archive an accurate classification, Vapnik and Lerner [53] proposed to choose the separating hyperplane that maximizes the margin; distance the hyperplane and the closest example. A hyperplane like that, if found, is likely to correctly classify *unseen* or the testing dataset.

If the datasets \mathbf{D} in the training set are mathematically expressed as follows:

$$\mathbf{D} = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_m, y_m)\},$$

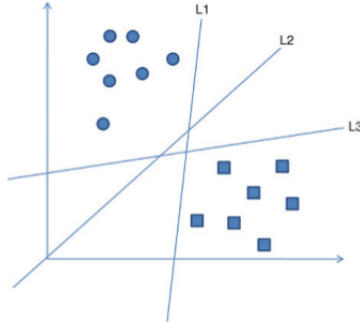


Figure 2.1: Hyperplanes separating data [1]

where \mathbf{x}_i is an n -dimensional real vector, y_i is either 1 or -1 denoting the class to which the point \mathbf{x}_i belongs. The SVM classification function $f(\mathbf{x})$ is

$$f(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} - b, \quad (2.1)$$

where \mathbf{w} is the weight vector and b is the bias.

In the binary classification case, to correctly classify the training set, $f(\mathbf{x})$ must return positive numbers for positive class data and negative numbers otherwise, for every point \mathbf{x}_i in \mathbf{D} ,

$$f(\mathbf{x}) = \begin{cases} \mathbf{w} \cdot \mathbf{x}_i - b > 0 & \text{if } y_i = 1, \text{ and} \\ \mathbf{w} \cdot \mathbf{x}_i - b < 0 & \text{if } y_i = -1 \end{cases}.$$

The conditions are restated as follows.

$$y_i (\mathbf{w} \cdot \mathbf{x}_i - b) > 0, \quad \forall (\mathbf{x}_i, y_i) \in \mathbf{D}. \quad (2.2)$$

As shown in Fig. 2.1, there are many hyperplanes ($f(\mathbf{x})$) that can be drawn. If there exists a linear function f that correctly classifies every point in \mathbf{D} is called linearly separable [1]. In this case, (2.2) can be written as

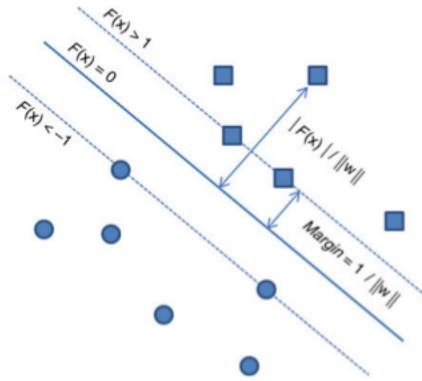


Figure 2.2: Maximizing the margin [1]

$$y_i (\mathbf{w} \cdot \mathbf{x}_i - b) \geq 1, \forall (\mathbf{x}_i, y_i) \in \mathbf{D}. \quad (2.3)$$

Note that (2.3) includes the equality sign, and the right side becomes 1 instead of 0. If \mathbf{D} is linearly separable (every point in \mathbf{D} satisfies (2.1)) then there exists an $f(\mathbf{x})$ that satisfies (2.3). This is because if there exist \mathbf{w} and b that satisfy (2.3), they can always be rescaled to satisfy (2.3). The distance from the hyperplane to a vector \mathbf{x}_i is formulated as $\frac{|f(\mathbf{x}_i)|}{\|\mathbf{w}\|}$. It is desired to obtain the hyperplane that maximizes the geometric distance to the closest datasets. Thus, the margin becomes

$$margin = \frac{|1|}{\|\mathbf{w}\|}$$

because when \mathbf{x}_i are the closest vectors, $f(\mathbf{x}) = 1$ (2.3). The closest vectors that satisfy (2.3) with the equality sign are called *support vectors*.

Maximizing the margin becomes minimizing $\|\mathbf{w}\|$. The training problem in an SVM becomes a **constrained optimization problem** as follows:

$$\begin{aligned}
\min_{\mathbf{w}, b} \quad & f(\mathbf{w}) = \frac{1}{2} \|\mathbf{w}\|^2, \\
\text{s.t.} \quad & y_i (\mathbf{w} \cdot \mathbf{x}_i - b) \geq 1 \quad \forall (\mathbf{x}_i, y_i) \in \mathbf{D}.
\end{aligned} \tag{2.4}$$

SVM Constrained Optimization Problem

The constrained optimization problem (2.4) is called a **primal problem** [1, 54]. It is characterized as follows:

- The objective function (2.4) is a convex function of \mathbf{w} .
- The constraints are linear in \mathbf{w} .

We can solve the constrained optimization problem using Lagrange multiplier methods [55]. Given a constrained optimization problem, we can construct another problem, called **dual problem**. The dual problem has the same optimal value as the primal problem, but with Lagrange multipliers providing the optimal solution. Primal optimization of nonlinear SVM has also been used [33, 56, 57].

The Lagrange function is

$$\mathcal{L}(\mathbf{w}, b, \boldsymbol{\alpha}) = \frac{1}{2} \|\mathbf{w}\|^2 - \sum_{i=1}^m \alpha_i y_i (\mathbf{w}^T \mathbf{x}_i + b - 1),$$

where the nonnegative auxiliary variables α are called Lagrange multipliers. The solution to the constrained optimization problem is determined by the saddle point of the Lagrange function $\mathcal{L}(\mathbf{w}, b, \boldsymbol{\alpha})$, which must be minimized with respect to \mathbf{w} and b . Thus, differentiating $\mathcal{L}(\mathbf{w}, b, \boldsymbol{\alpha})$ with respect to \mathbf{w} and b and setting the results equal to zero, we obtain the following two optimality conditions:

The derivatives

$$\frac{\partial \mathcal{L}(\mathbf{w}, b, \boldsymbol{\alpha})}{\partial \mathbf{w}} = \mathbf{w} - \sum_{i=1}^m \alpha_i y_i \mathbf{x}_i = \mathbf{0}, \quad \Rightarrow \mathbf{w} = \sum_{i=1}^m \alpha_i y_i \mathbf{x}_i, \quad (2.5)$$

$$\frac{\partial \mathcal{L}(\mathbf{w}, b, \boldsymbol{\alpha})}{\partial b} = - \sum_{i=1}^m \alpha_i y_i = 0, \quad \Rightarrow \sum_{i=1}^m \alpha_i y_i = 0. \quad (2.6)$$

The solution vector \mathbf{w} is defined in terms of an expansion that involves the m training examples.

The dual problem is as follows.

$$\begin{aligned} \max_{\boldsymbol{\alpha}} \quad & \mathcal{L}(\boldsymbol{\alpha}) = \sum_{i=1}^m \alpha_i - \frac{1}{2} \sum_{i=1}^m \sum_{j=1}^m \alpha_i \alpha_j y_i y_j (\mathbf{x}_i^T \mathbf{x}_j). \\ \text{s.t.} \quad & \boldsymbol{\alpha} \geq 0, \\ & \sum_{i=1}^m y_i \alpha_i = 0. \end{aligned} \quad (2.7)$$

The dual problem is entirely in terms of the training dataset. The function $\mathcal{L}(\boldsymbol{\alpha})$ to be maximized depends only on the input patterns in the form of a set of dot products $\{\mathbf{x}_i \cdot \mathbf{x}_j\}_{(i,j)=1}^m$.

Having determined the optimum Lagrange multipliers, denoted by α_i^* , the optimum weight vector \mathbf{w}^* can be calculated using (2.5).

$$\mathbf{w}^* = \sum_i \alpha_i^* y_i \mathbf{x}_i. \quad (2.8)$$

According to the property of the Karush-Kuhn-Tucker (KKT) conditions of optimization theory [1], the solution of the dual problem α_i^* must satisfy the following condition:

$$\alpha_i^* \{y_i(\mathbf{w}^T \mathbf{x}_i - b) - 1\} = 0 \quad \text{for } i = 1, 2, \dots, m \quad (2.9)$$

and either α_i^* or its corresponding constraint $y_i(\mathbf{w}^T \mathbf{x}_i - b) - 1$ must be nonzero. The optimal variables α_i^* will be nonzero (or nonnegative from (2.7)) when x_i is a support vector or $y_i(\mathbf{w}^T \mathbf{x}_i - b) - 1$. \mathbf{x}_i whose corresponding coefficients α_i are zero will not affect the optimal weight vector \mathbf{w}^* due to (2.8)[1]. The optimal weight vector \mathbf{w}^* will depend only on the support vectors, whose coefficients are positive.

The bias b is computed using the support vector \mathbf{x}_i :

$$b^* = \mathbf{w}^* \cdot \mathbf{x}_i - y_i.$$

The classification function $f(x)$ of (2.1) now becomes

$$f(\mathbf{x}) = \sum_i \alpha_i y_i \mathbf{x}_i \cdot \mathbf{x} - b. \quad (2.10)$$

Soft Margin - Linear nonseparable case

When the data are of a linear nonseparable case, the optimization problem (2.4) will not have a solution. A soft-margin SVM allows for a mislabeled dataset while still maximizing the margin [1]. The method introduces slack variables ξ_i , which measure the degree of misclassification.

Using a *slack variable* $\xi \geq 0$

$$\begin{cases} \mathbf{w}^T \mathbf{x}_i + b \geq 1 - \xi_i, & \text{for } y_i = 1, \\ \mathbf{w}^T \mathbf{x}_i + b \leq -1 + \xi_i, & \text{for } y_i = -1, \\ \xi_i \geq 0 & \forall i. \end{cases}$$

The primary problem is as follows.

$$\begin{aligned}
& \min_{\mathbf{w}, b, \boldsymbol{\xi}} \quad Q(\mathbf{w}, b, \boldsymbol{\xi}) = \frac{1}{2} \|\mathbf{w}\|^2 + C \sum_{i=1}^m \xi_i \\
& \text{s.t.} \quad y_i(\mathbf{w}^T \mathbf{x}_i + b) - 1 + \xi_i \geq 0, i = 1 \cdots m, \\
& \quad \quad \quad \xi_i \geq 0, \forall i.
\end{aligned}$$

The Lagrangian and its derivatives are

$$\begin{aligned}
\mathcal{L}(\mathbf{w}, b, \boldsymbol{\alpha}, \boldsymbol{\mu}) &= \frac{1}{2} \|\mathbf{w}\|^2 + C \sum_{i=1}^m \xi_i - \sum_{i=1}^m \alpha_i (y_i(\mathbf{w} \cdot \mathbf{x}_i - b) - 1 + \xi_i) - \sum_{i=1}^m \mu_i \xi_i, \\
\frac{\partial \mathcal{L}(\mathbf{w}, b, \boldsymbol{\alpha}, \boldsymbol{\mu})}{\partial \mathbf{w}} &= \mathbf{w} - \sum_{i=1}^m \alpha_i y_i \mathbf{x}_i = \mathbf{0}, \Rightarrow \mathbf{w} = \sum_{i=1}^m \alpha_i y_i \mathbf{x}_i, \\
\frac{\partial \mathcal{L}(\mathbf{w}, b, \boldsymbol{\alpha}, \boldsymbol{\mu})}{\partial b} &= - \sum_{i=1}^m \alpha_i y_i = 0, \Rightarrow \sum_{i=1}^m \alpha_i y_i = 0, \\
C - \alpha_i - \mu_i &= 0.
\end{aligned}$$

The dual problem is as follows.

$$\begin{aligned}
\max_{\boldsymbol{\alpha}} \quad Q(\boldsymbol{\alpha}) &= \sum_{i=1}^m \alpha_i - \frac{1}{2} \sum_{i=1}^m \sum_{j=1}^m \alpha_i \alpha_j y_i y_j (\mathbf{x}_i^T \mathbf{x}_j) \\
\text{s.t.} \quad 0 &\leq \alpha_i \leq C, \\
\boldsymbol{\alpha}^T \mathbf{y} &= 0.
\end{aligned} \tag{2.11}$$

Neither the slack variables ξ_i nor their Lagrange multipliers appear in the dual problem. The dual problem for the case of nonseparable patterns is similar to that for the simple case of linearly separable patterns except for a minor but important difference. The objective function $Q(\boldsymbol{\alpha})$ to be maximized is the same in both cases. The nonseparable case differs from the separable case in that the constraint $0 \leq \alpha_i$ is replaced with the more stringent constraint $0 \leq \alpha_i \leq C$. Except for this modification, the constrained optimization for the nonseparable case and the computations of the optimum values of the weight vector \mathbf{w} and bias b proceed in the same way as in the linearly separable case.

Using the KKT conditions defined by

$$\begin{aligned}
\alpha_i^* \{y_i(\mathbf{w}^T \mathbf{x}_i - b) - 1 + \xi_i\} &= 0 \quad \text{for } i = 1, 2, \dots, m. \\
\mu_i \xi_i &= 0 \quad \text{for } i = 1, 2, \dots, m.
\end{aligned} \tag{2.12}$$

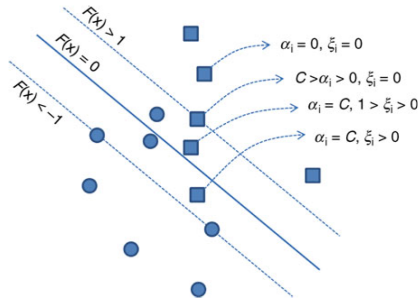


Figure 2.3: Linear nonseparable case [1]

Eq. (2.12) is a rewrite of (2.9) except for the replacement of the unity term $(1 - \xi_i)$. μ_i are Lagrange multipliers that have been introduced to enforce the nonnegativity of the slack variables x_i for all i . At the saddle point, the derivative of the Lagrange function for the primal problem with respect to the slack variable x_i is zero, the evaluation of which yields

$$\alpha_i + \mu_i = C. \quad (2.13)$$

Combining (2.12) and (2.13),

$$\begin{aligned} \xi_i &= 0, \text{ if } \alpha_i < C, \\ \xi_i &\geq 0, \text{ if } \alpha_i = C. \end{aligned}$$

We can graphically display the relationships between α_i , ξ_i , and C in Fig. 2.3.

Datasets outside the margin will have $\alpha_i = 0$ and $\xi_i = 0$ and those on the margin line will have $C > \alpha_i > 0$ and still $\xi_i = 0$. Datasets within the margin will have $\alpha_i = C$. Among them, those correctly classified will have $0 < \xi_i < 1$ and those misclassified points will have $\xi_i > 1$.

Calculating b can be done by taking advantage of the KKT conditions in Eq. (2.12) [58, 59]. These KKT conditions imply that at the point of the solution, the product between the dual variables α and the constraints must be equal to zero.

Kernel Trick for Nonlinear Classification

If the training data are not linearly separable, there is no linear hyperplane that can separate the classes. To learn a nonlinear function, linear SVM must be extended to nonlinear SVM for the classification of nonlinearly separable data. The process of finding classification functions using nonlinear SVM consists of two steps [1]. First, the input vectors are transformed into high-dimensional feature vectors where the training data can be linearly separated. Then, SVM are used to find the hyperplane of maximal margin in the new feature space. The separating hyperplane becomes a linear function in the transformed feature space but a nonlinear function in the original input space.

Let \mathbf{x} be a vector in the n -dimensional input space and $\varphi(\cdot)$ be a nonlinear mapping function from the input space to the high-dimensional feature space. The hyperplane representing the decision boundary in the feature space is defined as follows.

$$\mathbf{w} \cdot \varphi(\mathbf{x}) - b = 0,$$

where \mathbf{w} is a vector that denotes the hyperplane can map the training data in the high-dimensional feature space to the output space, and b is the bias. Using the $\varphi(\cdot)$ function, the weight becomes

$$\mathbf{w} = \sum_i \alpha_i y_i \varphi(\mathbf{x}_i).$$

The decision function of becomes

$$f(\mathbf{x}, \boldsymbol{\alpha}) = \sum_i^m \alpha_i y_i \varphi(\mathbf{x}_i) \cdot \varphi(\mathbf{x}) - b,$$

the dual problem of the soft-margin SVM (2.11) can be rewritten using the mapping function on the data vectors as follows:

$$f(\boldsymbol{\alpha}) = \sum_i \alpha_i - \frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y_i y_j \varphi(\mathbf{x}_i) \varphi(\mathbf{x}_j). \quad (2.14)$$

The calculation of the inner product in the transformed feature space seems quite complex and suffers from the curse of the dimensionality problem [1]. To avoid this problem, the kernel trick is used. The kernel trick replaces the inner product in the feature space with a kernel function K in the original input space as follows.

$$K(\mathbf{u}, \mathbf{v}) = \varphi(\mathbf{u}) \varphi(\mathbf{v}).$$

Mercers theorem proves that a kernel function K is valid if and only if the following conditions are satisfied, for any function $\psi(x)$ [20]:

$$\begin{aligned} \int K(\mathbf{u}, \mathbf{v}) \psi(\mathbf{u}) \psi(\mathbf{v}) d\mathbf{x} d\mathbf{y} &\leq 0 \\ \text{where } \int \psi(\mathbf{x})^2 d\mathbf{x} &\leq 0. \end{aligned}$$

Mercers theorem ensures that the kernel function can always be expressed as the inner product between pairs of input vectors in some high-dimensional space. Therefore, the inner product can be calculated using the kernel function only with input vectors in the original space without transforming the input vectors into high-dimensional feature vectors.

The dual problem is now defined using the kernel function as follows:

$$\begin{aligned} \min_{\boldsymbol{\alpha}} \quad & f(\boldsymbol{\alpha}) = \frac{1}{2} \sum_{i=1}^m \sum_{j=1}^n \alpha_i \alpha_j y_i y_j K(\mathbf{x}_i, \mathbf{x}_j) - \sum_{i=1}^m \alpha_i \\ \text{s.t.} \quad & c_i(\boldsymbol{\alpha}) = \alpha_i \geq 0, \quad i = 1 \cdots m, \\ & c_{m+i}(\boldsymbol{\alpha}) = C - \alpha_i \geq 0, \quad i = 1 \cdots m. \end{aligned} \quad (2.15)$$

$K(\cdot, \cdot)$ is computed in the input space and no feature transformation will be performed, or no $\varphi(\cdot)$ will be calculated. The kernel function is a kind of similarity function between two vectors where the output of the function is maximized when the two vectors become equivalent [1]. SVM can learn a function from any shape of data beyond vectors (such as trees or graphs) as long as a similarity function can be computed between any pair of data objects.

The derivative of Eq. (2.15) is

$$\nabla f(\boldsymbol{\alpha}) = \mathbf{Q}\boldsymbol{\alpha} - \mathbf{e}, \quad (2.16)$$

where matrix \mathbf{Q} ,

$$\mathbf{Q} = y_i y_j K(\mathbf{x}_i, \mathbf{x}_j). \quad (2.17)$$

Examples of Kernels

1. Radial basis function kernel

$$K(\mathbf{x}_i, \mathbf{x}_j) = \exp(\gamma \|\mathbf{x}_i - \mathbf{x}_j\|^2).$$

2. Scalar product Kernel

$$K(\mathbf{x}_i, \mathbf{x}_j) = \langle \mathbf{x}_i, \mathbf{x}_j \rangle.$$

3. Polynomial Kernel

$$K(\mathbf{x}_i, \mathbf{x}_j) = (\mathbf{x}_i \cdot \mathbf{x}_j + 1)^d.$$

4. Sigmoid

$$K(\mathbf{x}_i, \mathbf{x}_j) = \tanh(\kappa \mathbf{x}_i \cdot \mathbf{x}_j + \rho).$$

Computation of the kernel values

Although the calculation of the n^2 components of the kernel matrix $K_{ij} = K(\mathbf{x}_i, \mathbf{x}_j)$ seems to be a simple quadratic complexity task, a more detailed analysis reveals a much more complicated picture [9].

- **Computing kernels is expensive** - Computing each kernel value involves computing the similarity between large data that represent the features. The calculation of kernel values often accounts for a significant amount of the total training time.
- **Computing the full kernel matrix is wasteful** - The expression of the gradient (Eq. (2.16)) depends only on the kernel values K_{ij} that involve at least one support vector (the other kernel values are multiplied by zero). The remaining kernel values that belong to the nonsupport vector have no impact on the solution. To determine which kernel values are actually needed, efficient SVM solvers compute no more than 15% to 50% additional kernel values[9].
- **The kernel matrix does not fit in memory** - When the number of examples grows, the kernel matrix K_{ij} becomes very large and cannot be stored in memory. Kernel values must be computed on the fly or retrieved from a cache of often accessed values[22]. The cache hit rate of the cache values of the kernel matrix becomes an important factor in how long the training time is[9].

Summary

In this chapter, we have described the SVM supervised learning technique. We have a summary of the SVM solvers used for training SVM. We describe the kernel trick for nonlinear classification. We describe some of the challenges of training Support Vector Machines. The

optimization techniques described in the next chapter can be used to find a solution to the SVM quadratic problem.

Chapter 3: SVM Quadratic Programming problem

SVM Quadratic Programming problem

A SVM Quadratic Programming problem (2.15) is

$$\begin{aligned} \min_{\alpha} \quad & f(\alpha) \\ \text{s.t.} \quad & g_i(\alpha) = 0, \quad i = 1, \dots, q, \\ & c_j(\alpha) \geq 0, \quad j = 1, \dots, p \end{aligned} \tag{3.1}$$

for $\alpha \in \mathbb{R}^n$, $f, g_i, c_j : \mathbb{R}^n \rightarrow \mathbb{R}$, and f is quadratic and g_i, c_j are linear functions. $g_i(\alpha) = 0, \quad i = 1, \dots, q$ is the equality constraint and $c_j(\alpha) \geq 0, \quad j = 1, \dots, p$, the inequality constraint. Griva [54], Boyd [60], and Wright [61] provide details on different approaches for solving problems with inequalities or equalities constraints. I will discuss the Augmented Lagrangian Method Fast Projected Gradient Method ALFPGM and Nonlinear Rescaling Augmented Lagrangian (NRAL) technique.

Augmented Lagrangian Method

An Augmented Lagrangian Method [62, 63] iteratively minimizes an Augmented Lagrangian function with respect to its primal variables and then updates its dual variables. Augmented Lagrangian function similarities to penalty methods [54, 64] in that they replace a constrained optimization problem by a series of unconstrained problems and add a penalty term to the objective; the difference is that the augmented Lagrangian method adds yet another term, designed to mimic a Lagrange multiplier. The unconstrained objective is the

Lagrangian of the constrained problem, with an additional penalty term (the augmentation). Iteration continues until the first-order optimality condition is satisfied. An example of an Augmented Lagrangian function is

$$A_\mu(\boldsymbol{\alpha}, \lambda) = f(\boldsymbol{\alpha}) - \lambda^T g(\boldsymbol{\alpha}) + \frac{1}{2} \mu g(\boldsymbol{\alpha})^T g(\boldsymbol{\alpha}),$$

where A_μ is a Lagrangian with a penalty term.

An Augmented Lagrangian method iterates

$$\begin{aligned} \boldsymbol{\alpha}_{s+1} &= \arg \min_{\boldsymbol{\alpha} \in \mathbb{R}^m} A_\mu(\boldsymbol{\alpha}, \lambda_s), \\ \lambda_{s+1} &= \lambda_s - \mu g(\boldsymbol{\alpha}_s) \end{aligned} \tag{3.2}$$

over s until $\nabla A_\mu(\boldsymbol{\alpha}_s, \lambda_s) \approx 0$. Bertsekas describes Augmented Lagrangian methods in his book [65]. Conn et al. [66] addressed computational issues related to Augmented Lagrangian techniques using an early version of the LANCELOT solver. Augmented Lagrangian techniques avoid ill-conditioning, but they perform an unconstrained minimization at each step, which can lead to using a lot of steps to arrive at the solution. The solution to $\boldsymbol{\alpha}_{s+1} \approx \arg \min_{\boldsymbol{\alpha} \in \mathbb{R}^m} A_\mu(\boldsymbol{\alpha}, \lambda_s)$ can be found using first order methods like the Fast Projected Gradient Method.

The parameter μ should be large enough so that the augmented Lagrangian function has a local minimizer in $\boldsymbol{\alpha}$. If μ is too small, then the unconstrained subproblem may not have a solution [54].

Gradient Projection Methods

To find the solution to $\boldsymbol{\alpha}_{s+1} \approx \arg \min_{\boldsymbol{\alpha} \in \mathbb{R}^m} A_\mu(\boldsymbol{\alpha}, \lambda_s)$ in Eq. (3.2), first order like gradient projection methods like the ISTA (Iterative Shrinkage-Thresholding Algorithm) and FISTA (Fast Iterative Shrinkage-Thresholding Algorithm) [45, 67] methods minimize functions of

the form

$$F(\boldsymbol{\alpha}) = f(\boldsymbol{\alpha}) + g(\boldsymbol{\alpha}), \quad (3.3)$$

where

$$\boldsymbol{\alpha} \in R^N,$$

$f : R^N \rightarrow R$: smooth, convex continuously differentiable with Lipschitz continuous gradient $L(f)$: $\|\nabla f(\boldsymbol{\alpha}) - \nabla f(\bar{\boldsymbol{\alpha}})\|_2 \leq L(f)\|\boldsymbol{\alpha} - \bar{\boldsymbol{\alpha}}\|_2$ for every $\boldsymbol{\alpha}, \bar{\boldsymbol{\alpha}} \in R^N$ where $L(f) > 0$ is the Lipschitz constant for $\nabla f(\boldsymbol{\alpha})$ and

$g : R^N \rightarrow R$: continuous, convex, but possibly non-smooth function.

The basic idea of the iterative shrinkage algorithm is to build at each iteration a regularization of the differentiable linearized function part in the objective [45]. In each iteration, these algorithms calculate

$$\alpha_i = p_L(\alpha_{i-1} - t_i \nabla f(\boldsymbol{\alpha})), \quad (3.4)$$

where

$$p_L(\boldsymbol{\alpha}) = \arg \min_{\boldsymbol{\alpha}} F(\boldsymbol{\alpha}) \quad (3.5)$$

and t_i is the step-size parameter. From definition of proximity operator, when $g(\boldsymbol{\alpha}) = 0$ then the above Eq. (3.9) is given as

$$\alpha_i = \alpha_{i-1} - t_i \nabla f(\boldsymbol{\alpha}) \quad (3.6)$$

for minimizing the function with a Lipschitz continuous gradient $L(f)$. On the other hand, if $f(\alpha) = 0$, then Eq. ((3.4)) reduces to

$$\alpha_i = p_L(\alpha_{i-1}), \quad (3.7)$$

for minimizing non-differentiable function. Such schemes are called as *forward-backward proximal splitting* ([68]). with forward scheme as a gradient step using $f(\alpha)$ and the backward or implicit scheme using the function $g(\alpha)$. There are two variants of FISTA and ISTA; One with a fixed step size t and the other with varying step size t_i or backtracking. Previous works have compared the FISTA and ISTA. Fast Iterative-Shrinkage Thresholding Algorithm (FISTA) is the faster version of the ISTA algorithm that preserves the simplicity of ISTA. FISTA'S theoretical and practical rate of convergence of FISTA is significantly better than ISTA [45]

FISTA with fixed step length

Fast Iterative-Shrinkage Thresholding Algorithm (FISTA) is the faster version of ISTA algorithm which also preserves the simplicity of ISTA. FISTA'S theoretical and practical rate of convergence of FISTA is significantly better than that of ISTA [45].

Algorithm 1 FISTA with Constant Step size

Require: $\alpha_0 \in \mathbb{R}^N, L := L(f)$.

$k := 0$

$t := 1$

$\bar{\alpha} := \alpha_0$

repeat

$\alpha_k := p_L(y_k)$

$t_{k+1} := \frac{1 + \sqrt{1 + 4t_k^2}}{2}$

$\bar{\alpha}_{k+1} := \alpha_k + \left(\frac{t_k - 1}{t_{k+1}}\right)(\alpha_k - \alpha_{k-1})$

$k := k + 1$

until convergence.

The error in the k^{th} iteration with respect to the true minimum α_* is bounded as $F(\alpha_k) - F(\alpha_*) \leq O(1/(k+1)^2)$. FISTA algorithm converges with a convergence rate $O(1/k^2)$ as discussed in the classical work of [69] and Theorem 4.4 in [45].

MFISTA

The sequence of function values generated by FISTA is not necessarily nonincreasing. A monotone version of FISTA called MFISTA, which is a descent method and has the same convergence rate as FISTA. MFISTA necessitates the computation of the objective function $F(\cdot)$ which can be expensive.

Algorithm 2 MFISTA with constant Step size

Require: $\alpha_0 \in \mathbb{R}^N, L := L(f)$.

$k := 0$

$t := 1$

$\bar{\alpha} := \alpha_0$

repeat

$\mathbf{z}_k := \mathbf{p}_L(\mathbf{y}_k)$

$\alpha_k \in \mathbb{R}^N, F(\alpha_k) = \min(F(\mathbf{z}_k), F(\alpha_{k-1}))$

$t_{k+1} := \frac{1 + \sqrt{1 + 4t_k^2}}{2}$

$\bar{\alpha}_{k+1} := \alpha_k + \left(\frac{t_k}{t_{k+1}}\right)(\mathbf{z}_k - \alpha_k) + \left(\frac{t_k - 1}{t_{k+1}}\right)(\alpha_k - \alpha_{k-1})$

$k := k + 1$

until convergence.

Restart FISTA

A common observation when running an accelerated method is the appearance of ripples in the trace of the objective value. Restart FISTA was developed to address this issue and was shown to improve the convergence rate of the accelerated gradient [70]. It is a heuristic technique that restarts the algorithm whenever a condition on the objective function value is met.

Algorithm 3 Restart FISTA with constant step size

Require: $\alpha_0 \in \mathbb{R}^N, L := L(f)$.

$k := 0$

$t := 1$

$\bar{\alpha} := \alpha_0$

repeat

$\alpha_k := p_L(\mathbf{y}_k)$

$t_{k+1} := \frac{1 + \sqrt{1 + 4t_k^2}}{2}$

$\bar{\alpha}_{k+1} := \alpha_k + \left(\frac{t_k - 1}{t_{k+1}}\right)(\alpha_k - \alpha_{k-1})$

$k := k + 1$

 Restart if $(\bar{\alpha}_k - \alpha_{k+1})^T(\alpha_{k+1} - \alpha_k) \geq 0$, then $\bar{\alpha}_k = \alpha_k$.

until convergence.

RADA FISTA and GREEDY FISTA

In [67], Restarting and Adaptive FISTA (*RADA*) and GREEDY FISTA were proposed. They were designed to improve RESTART FISTA. Like RESTART FISTA, they do not require the computation of the objective function f for each iteration. According to [67], both offer a compromise between FISTA and MFISTA.

The stopping criterion for all of the above algorithms is given as

$$\left| \frac{f(\alpha_k) - f(\alpha_{k-1})}{f(\alpha_{k-1})} \right| < \epsilon, \quad (3.8)$$

where ϵ is some arbitrarily small quantity (usually $< 10^{-8}$).

Error at the k th iteration with respect to the true minimum α_* is bounded as $F(\alpha_k) -$

Algorithm 4 RADA with constant step size

Require: $\alpha_0 \in \mathbb{R}^N, L := L(f)$.

$k := 0$

$t := 1$

$\bar{\alpha} := \alpha_0$

repeat

$\alpha_k := p_L(\mathbf{y}_k)$

$t_{k+1} := \frac{1+\sqrt{1+4t_k^2}}{2}$

$\bar{\alpha}_{k+1} := \alpha_k + \left(\frac{t_k-1}{t_{k+1}}\right)(\alpha_k - \alpha_{k-1})$

 Restarting if $(\bar{\alpha}_k - \alpha_{k+1})^T(\alpha_{k+1} - \alpha_k) \geq 0$

$r = \eta r$ and $\bar{\alpha}_k = \alpha_k$

$r = \eta r, t_k = t_k$ and $\bar{\alpha}_k = \alpha_k$

$k := k + 1$

until convergence.

Algorithm 5 GREEDY FISTA with constant step size

Require: $\alpha_0 \in \mathbb{R}^N, L := L(f)$.

$k := 0$

$t := 1$

$\bar{\alpha} := \alpha_0$

repeat

$\alpha_k := p_L(\mathbf{y}_k)$

$t_{k+1} := \frac{1+\sqrt{1+4t_k^2}}{2}$

$\bar{\alpha}_{k+1} := \alpha_k + \left(\frac{t_k-1}{t_{k+1}}\right)(\alpha_k - \alpha_{k-1})$

 Restarting if $(\bar{\alpha}_k - \alpha_{k+1})^T(\alpha_{k+1} - \alpha_k) \geq 0$, then $\bar{\alpha}_k = \alpha_k$.

 Safeguard if $\|\alpha_{k+1} - \alpha_k\| \geq \|\alpha_1 - \alpha_0\|$, then $\mu = \max(\eta\mu, 1/L)$. $k := k + 1$

until convergence.

$$F(\boldsymbol{\alpha}_*) \leq O(1/k^2)[45].$$

Modified Barrier Method

To solve inequality-constrained optimization problems, Polyak introduced modified barrier methods [71]. Similar to Augmented Lagrangian methods, modified barrier methods explicitly use dual variables to avoid the ill-conditioning seen in classical barrier methods. The modified barrier method iteratively minimizes a modified barrier function with respect to its primal variables and then updates its dual variables. Iteration continues until the solution is found. An example of a modified barrier function is the logarithmic modified barrier function

$$\mathcal{L}_\mu(\boldsymbol{\alpha}, \boldsymbol{\nu}) = f(\boldsymbol{\alpha}) - \frac{1}{\mu} \sum_{i=1}^p \nu_i \log(\mu c_i(\boldsymbol{\alpha}) + 1).$$

The modified barrier method iterates

$$\begin{aligned} \boldsymbol{\alpha}_{s+1} &\approx \arg \min_{\boldsymbol{\alpha} \in \mathbb{R}^m} \mathcal{L}_\mu(\boldsymbol{\alpha}, \boldsymbol{\nu}_s), \\ (\boldsymbol{\nu}_{s+1})_i &= \frac{(\boldsymbol{\nu}_s)_i}{\mu c_i(\boldsymbol{\alpha}_{s+1}) + 1} \end{aligned}$$

over s until $\nabla \mathcal{L}(\boldsymbol{\alpha}_s, \boldsymbol{\nu}_s) \approx 0$ and $\boldsymbol{\nu}_i c_i(\boldsymbol{\alpha}) \approx 0$, $i = 1, \dots, p$. This method will converge for a fixed $\mu > 0$, so it avoids the ill-conditioning of the Hessian seen in classical barrier methods. One problem with the modified barrier method is that the modified barrier function is not defined for all real numbers, which can lead to numerical difficulties.

Nonlinear Rescaling Method

Polyak and Teboulle [12] introduced the nonlinear rescaling method as a generalization of the modified barrier method. The nonlinear rescaling method transforms the objective function and the constraints of a given constrained optimization problem into another problem which is equivalent to the original one. A nonlinear transformation parameterized by

a positive scalar parameter and based on a smooth sealing function is used to transform the constraints[12]. The method consists of sequential unconstrained minimization of the classical Lagrangian for the equivalent problem, followed by an update of the Lagrange multipliers. The nonlinear rescaling method transforms the constraints $c_i(\alpha)$ to an equivalent set using a class of functions with the following properties [12]:

$$\left. \begin{aligned} \psi(0) &= 0, \quad \psi'(t) > 0, \quad \psi'(0) = 1, \quad \psi''(t) < 0, \\ \psi'(t) &\leq \frac{a}{(t+1)} \\ -\psi''(t) &\leq \frac{b}{(t+1)^2} \end{aligned} \right\} \quad t \geq 0, a > 0, b > 0$$

For $\mu > 0$, the equivalent problem to (3.1) is

$$\begin{aligned} \min_{\boldsymbol{\alpha}} \quad & f(\boldsymbol{\alpha}) \\ \text{s.t.} \quad & \frac{1}{\mu} \psi(\mu c_i(\boldsymbol{\alpha})) \geq 0, \quad i = 1 \cdots p \end{aligned} \tag{3.9}$$

and the Lagrangian of the equivalent problem is

$$\mathcal{L}_\mu(\boldsymbol{\alpha}, \boldsymbol{\nu}) = f(\boldsymbol{\alpha}) - \frac{1}{\mu} \sum_{i=1}^p \nu_i \psi(\mu c_i(\boldsymbol{\alpha})).$$

Similar to the modified barrier method, the nonlinear rescaling method can solve (3.1) by minimizing the Lagrangian for the equivalent problem for fixed $\mu > 0$ and updating the multipliers $\boldsymbol{\nu}$. So the nonlinear rescaling method iterates

$$\begin{aligned} \boldsymbol{\alpha}_{s+1} &\approx \arg \min_{\boldsymbol{\alpha} \in \mathbb{R}^m} \mathcal{L}_\mu(\boldsymbol{\alpha}, \boldsymbol{\nu}_s), \\ (\boldsymbol{\nu}_{s+1})_i &= \psi'(\mu c_i(\boldsymbol{\alpha}_{s+1})) (\boldsymbol{\nu}_s)_i, \quad i = 1, \cdots, p \end{aligned}$$

over s until the solution is found. The nonlinear rescaling method will converge without increasing the scaling parameter μ , so it avoids the ill-conditioning seen in classical barrier methods.

Nonlinear Rescaling Augmented Lagrangian (NRAL) Technique

For problems with equality and inequality constraints, Griva and Polyak [13, 72] introduced the Nonlinear Rescaling Augmented Lagrangian (NRAL) technique. The NRAL technique combines the nonlinear rescaling method for inequality constraints [12, 71] with the Augmented Lagrangian method for equality constraints [62, 63]. The NRAL technique is a multiplier-based method that avoids the ill-conditioning seen in classical penalty and barrier methods by exploiting dual variables, or multipliers, as a driving force for the convergence of the methods to the solution.

The NRAL technique solves the problem (3.1) by constructing an equivalent problem in which the constraints are rescaled by a function $\psi : -\infty \leq t_0 < t < t_1 \leq +\infty$ such that $\psi(t)$ has the following properties:

$$\left. \begin{aligned} \psi(0) &= 0, \quad \psi'(t) > 0, \quad \psi'(0) = 1, \quad \psi''(t) < 0, \\ \psi'(t) &\leq \frac{a}{(t+1)} \\ -\psi''(t) &\leq \frac{b}{(t+1)^2} \end{aligned} \right\} \quad t \geq 0, a > 0, b > 0.$$

For $\mu > 0$, the following problem is equivalent to the original problem:

$$\begin{aligned} \min_{\boldsymbol{\alpha}} \quad & f(\boldsymbol{\alpha}) \\ \text{s.t.} \quad & g_i(\boldsymbol{\alpha}) = 0, \quad i = 1, \dots, q, \\ & \frac{1}{\mu} \psi(\mu c_j(\boldsymbol{\alpha})) \geq 0, \quad j = 1, \dots, p. \end{aligned} \tag{3.10}$$

In [73] the transformation ψ was chosen as

$$\psi(t) = \begin{cases} \log(t+1) & t > -0.5 \\ -2t^2 + \log(.5) + .5 & t \leq -0.5 \end{cases}$$

Griva and Polyak [72] actively used this transformation for solving nonlinear optimization problems using the nonlinear rescaling method. The Augmented Lagrangian for this equivalent problem is

$$\mathcal{L}_\mu(\boldsymbol{\alpha}, \lambda, \boldsymbol{\nu}) = f(\boldsymbol{\alpha}) - \lambda^T g(\boldsymbol{\alpha}) - \frac{1}{\mu} \sum_{i=1}^p \nu_i \psi(\mu c_i(\boldsymbol{\alpha})) + \frac{\mu}{2} \sum_{j=1}^q (g_j(\boldsymbol{\alpha}))^2,$$

where and $\mu > 0$ is the scaling parameter, $\lambda \in \mathbb{R}^q$ and $\boldsymbol{\nu} \in \mathbb{R}^p$ are dual variables.

As a multiplier-based method, the NRAL technique minimizes the Augmented Lagrangian for the equivalent problem with respect to the primal variables, and then updates the dual variables, or multipliers. If $(\boldsymbol{\alpha}^s, \lambda^s, \boldsymbol{\nu}^s)$ is a current iterate of the NRAL technique, then the next iterate $(\boldsymbol{\alpha}^{s+1}, \lambda^{s+1}, \boldsymbol{\nu}^{s+1})$ is found by solving the unconstrained minimization problem.

$$\boldsymbol{\alpha}^{s+1} = \arg \min_{\boldsymbol{\alpha} \in \mathbb{R}^n} \mathcal{L}_\mu(\boldsymbol{\alpha}^s, \lambda^s, \boldsymbol{\nu}^s)$$

and updating the multipliers;

$$\begin{aligned} \nu_i^{s+1} &= \nu_i^s \psi'(\mu c_i(\boldsymbol{\alpha}^{s+1})), \quad i = 1, \dots, p, \\ \lambda_j^{s+1} &= \lambda_j^s - \mu g_j(\boldsymbol{\alpha}^{s+1}), \quad j = 1, \dots, q. \end{aligned}$$

At each iteration of the NRAL technique, any unconstrained minimization routine can be used to find $\arg \min_{\boldsymbol{\alpha} \in \mathbb{R}^m} \mathcal{L}_\mu(\boldsymbol{\alpha}^s, \lambda^s, \boldsymbol{\nu}^s)$. If $(\boldsymbol{\alpha}_*, \lambda_*, \boldsymbol{\nu}_*)$ is the solution, then for μ large

enough, under second-order optimality conditions, the iterates $(\boldsymbol{\alpha}^s, \lambda^s, \boldsymbol{\nu}^s)$ derived using the NRAL technique will converge to the solution linearly.

Although the NRAL technique can avoid the ill-conditioning seen with penalty and barrier methods, since it converges for a fixed bounded scaling parameter, it still requires solving a minimization problem at each iteration, which can be computationally expensive. Also, increasing the scaling parameter at each iteration can help convergence, but ill-conditioning may occur if the scaling parameter becomes too large.

Summary

In this chapter, we have described the use of Lagrange multiplier methods to solve the SVM quadratic programming problem. We described the Augmented Lagrangian Fast Projected Gradient Method (ALFPGM). We described various Gradient Projection Methods or FISTA. We described the Nonlinear Rescaling Augmented Lagrangian (NRAL) technique. The next chapter will address the problem of solving the SVM problem for large data set problem where the kernel matrix is large.

Chapter 4: Working set selection method for SVM

Decomposition

The SVM problem with the kernel trick is

$$\begin{aligned} \max_{\boldsymbol{\alpha}} \quad & f(\boldsymbol{\alpha}) = \sum_{i=1}^m \alpha_i^* - \frac{1}{2} \sum_{i=1, j=1}^m y_i \alpha_i^* y_j \alpha_j^* K(\mathbf{x}_i, \mathbf{x}_j) \\ \text{s.t.} \quad & \alpha_i^* = \alpha_i, \\ & 0 \leq \alpha_i^* \leq C, \quad i = 1, \dots, m, \\ & \sum_i y_i \alpha_i^* = 0. \end{aligned} \tag{4.1}$$

The $m \times m$ kernel matrix K in Eq. (4.1) is often dense and large; thus, a memory shortage prohibits the formation of Q and keeping it in memory, thus making it difficult to solve the SVM problem when the number of training data points is large. Decomposition methods [23, 74–76] based on selection of working sets play an important role in training a large-scale SVM. These methods consider only a small subset of data per iteration, which requires less memory usage.

A subset B of the dataset called the working set is chosen to be optimized while the rest of the variables remain fixed. Instead of updating all the coefficients of the vector $\boldsymbol{\alpha}$, each iteration of the decomposition method optimizes a subset of the coefficients $\alpha_i, i \in B$ and leaves the remaining coefficients $\alpha_j, j \notin B$ unchanged.

The SVM problem with the selected working set B becomes

$$\begin{aligned}
\max_{\boldsymbol{\alpha}} \quad & f(\boldsymbol{\alpha}) = \sum_{i=1}^n \alpha_i^* - \frac{1}{2} \sum_{i=1, j=1}^n y_i \alpha_i^* y_j \alpha_j^* K(\mathbf{x}_i, \mathbf{x}_j) \\
\text{s.t.} \quad & \forall_i \notin B \quad \alpha_i^* = \alpha_i, \\
& \forall_i \in B \quad 0 \leq \alpha_i^* \leq C, \quad i = 1, \dots, m, \\
& \sum_i y_i \alpha_i^* = 0.
\end{aligned} \tag{4.2}$$

We can define a matrix $\mathbf{Q} = y_i y_j K(\mathbf{x}_i, \mathbf{x}_j)$, $i = 1, \dots, n$, $j = 1, \dots, n$.

Eq. (4.2) becomes

$$\begin{aligned}
\max_{\boldsymbol{\alpha}} \quad & f(\boldsymbol{\alpha}) = \sum_{i=1}^n \alpha_i^* - \frac{1}{2} \alpha_i^* \mathbf{Q} \alpha_j^* \\
\text{s.t.} \quad & \forall_i \notin B \quad \alpha_i^* = \alpha_i, \\
& \forall_i \in B \quad 0 \leq \alpha_i^* \leq C, \quad i = 1, \dots, m, \\
& \sum_i y_i \alpha_i^* = 0.
\end{aligned} \tag{4.3}$$

Starting from a coefficient vector $\boldsymbol{\alpha}$ we can compute a new coefficient vector $\boldsymbol{\alpha}^*$ by adding an additional constraint to the dual problem (2.15) that represents the frozen coefficients. We can rewrite (4.2) as a quadratic programming problem in variables $\alpha_{i,i \in B}$ and remove additive terms that do not involve optimization variables $\boldsymbol{\alpha}^*$:

$$\begin{aligned}
\max_{\alpha} \quad & \sum_{i \in B} \alpha_i^* \left(1 - y_i \sum_{j \notin B}^n y_j \alpha_j^* K(\mathbf{x}_i, \mathbf{x}_j) \right) - \frac{1}{2} \sum_{i \in B} \sum_{j \in B} y_i \alpha_i^* y_j \alpha_j^* K(\mathbf{x}_i, \mathbf{x}_j) \\
\text{s.t.} \quad & \sum_{i \in B} y_i \alpha_i^* = - \sum_{j \notin B} y_j \alpha_j^*, \\
& \forall i \in B \quad 0 \leq \alpha_i^* \leq C, \quad i = 1, \dots, m.
\end{aligned} \tag{4.4}$$

Each iteration selects a working set and solves the corresponding sub-problem using any suitable optimization algorithm. The gradient is efficiently updated by evaluating the difference between the old gradient and new gradient and updates the variables. The algorithm stops when it achieves the optimality criterion. All these operations can be achieved using only the kernel matrix rows whose indices are in B .

The definition of (4.2) ensures that $\mathbf{F}(\alpha^*) \geq \mathbf{F}(\alpha)$. The question is then to define a working set selection scheme that ensures that the increasing values of the dual reach the maximum.

Since each iteration involves only B columns of the matrix Q , the decomposition methods use the operating memory economically [75]. The algorithm repeats the **select working set then optimize** process until the global optimality conditions are satisfied. While B denotes the working set with variables l , N denotes the non-working set with variables $(m - l)$. Then α , y , and Q can be written as:

$$\alpha = \begin{bmatrix} \alpha_B \\ \alpha_N \end{bmatrix}, \quad y = \begin{bmatrix} y_B \\ y_N \end{bmatrix}, \quad Q = \begin{bmatrix} Q_{BB} & Q_{BN} \\ Q_{NB} & Q_{NN} \end{bmatrix}, \quad Q_{ij} = y_i K_{ij} y_j.$$

The optimization subproblem can be rewritten.

$$\begin{aligned}
& \min_{\alpha_B} \quad \frac{1}{2} \begin{bmatrix} \alpha_B^T & \alpha_N^T \end{bmatrix} \begin{bmatrix} \mathbf{Q}_{BB} & \mathbf{Q}_{BN} \\ \mathbf{Q}_{NB} & \mathbf{Q}_{NN} \end{bmatrix} \begin{bmatrix} \alpha_B \\ \alpha_N \end{bmatrix} - \begin{bmatrix} \mathbf{e}_B^T & \mathbf{e}_N^T \end{bmatrix} \begin{bmatrix} \alpha_B \\ \alpha_N \end{bmatrix} \\
& \text{s.t.} \quad \alpha_B^T y_B + \alpha_N^T y_N = 0, \\
& \quad \quad 0 \leq \alpha_B \leq C
\end{aligned} \tag{4.5}$$

with a fixed α_N , this is the same as

$$\begin{aligned}
& \min_{\alpha_B} \quad \frac{1}{2} \alpha_B^T \mathbf{Q}_{BB} \alpha_B + \alpha_B^T \chi \\
& \text{s.t.} \quad \alpha_B^T y_B + G_k = 0, \\
& \quad \quad 0 \leq \alpha_B \leq C,
\end{aligned} \tag{4.6}$$

where $\chi = \mathbf{Q}_{BN} \alpha_N - \mathbf{e}_B$ and $G_k = \alpha_N^T y_N$.

The reduced problem (4.5) can be solved much faster than the original problem (4.2).

The resulting algorithm is shown as Algorithm 4.

Algorithm 6 Decomposition method

- 1: **while** global minimum not reached **do**
 - 2: find working set B
 - 3: find α_B for working set B using a quadratic problem solver.
 - 4: Update the gradient
 - 5: Update α with α_B
-

Working set selection methods

The selection of the working set is critical to the success of the decomposition technique.

Below we describe a working set selection (WSS) method that was developed by Keerthi et al. [18, 21] called the maximum violating pair method.

Working set selection $WSS - MVP$

The KKT condition of (4.6) shows that there is a scalar b such that

$$\begin{aligned}
y_t = 1, \alpha_t < C &\Rightarrow (Q\alpha - \mathbf{e})_t + b \geq 0 \Rightarrow b \geq -(Q\alpha + \mathbf{e})_t = -\nabla f(\alpha)_t, \\
y_t = -1, \alpha_t > 0 &\Rightarrow (Q\alpha - \mathbf{e})_t - b \leq 0 \Rightarrow b \geq (Q\alpha + \mathbf{e})_t = \nabla f(\alpha)_t, \\
y_t = -1, \alpha_t < C &\Rightarrow (Q\alpha - \mathbf{e})_t - b \geq 0 \Rightarrow b \leq (Q\alpha + \mathbf{e})_t = \nabla f(\alpha)_t, \\
y_t = 1, \alpha_t > 0 &\Rightarrow (Q\alpha - \mathbf{e})_t + b \leq 0 \Rightarrow b \leq -(Q\alpha + \mathbf{e})_t = -\nabla f(\alpha)_t,
\end{aligned} \tag{4.7}$$

where $f(\alpha) \equiv \frac{1}{2}\alpha^T Q\alpha - e^T \alpha$ and $\nabla f(\alpha)$ is the gradient of $f(\alpha)$.

This can be rewritten as

$$\begin{aligned}
\nabla f(\alpha) + by_t &\geq 0, \quad \text{if } \alpha_t < C, \\
\nabla f(\alpha) + by_t &\leq 0, \quad \text{if } \alpha_t > 0.
\end{aligned}$$

Alternatively, it can be stated that a vector α is a stationary point of (4.7) if and only if there is a number b and two nonnegative vectors ρ and θ such that

$$\begin{aligned}
\nabla f(\alpha) + by_t &= \rho - \theta, \\
\rho_t \alpha_t &= 0, \quad \theta_t (C - \alpha_t) = 0, \quad \rho_t \geq 0, \quad \theta_t \geq 0, \quad t = 1, \dots, m.
\end{aligned}$$

Consider the sets:

$$\begin{aligned}
I_{up}(\alpha) &\equiv \{t \mid y_t = 1, \alpha_t < C \text{ or } y_t = -1, \alpha_t > 0\}, \\
I_{low}(\alpha) &\equiv \{t \mid y_t = -1, \alpha_t < C \text{ or } y_t = 1, \alpha_t > 0\}.
\end{aligned}$$

Then we have the following.

$$\begin{aligned} i &\in \arg \max_{t \in I_{up}(\alpha)} \{-y_t \nabla f(\alpha)_t\}, \\ j &\in \arg \min_{t \in I_{low}(\alpha)} \{-y_t \nabla f(\alpha)_t\}. \end{aligned} \tag{4.8}$$

The Most Violating Pair (MVP) algorithm selects a pair with i and j from (4.8). The MVP unfortunately does not converge for all cases in practice. Fan et al. [23] improved on the MVP which by using second order information. This is shown in Algorithm 8.

Algorithm 7 WSS_{MVP} Most Violating Pair

```

1:  $p$  = number of pairs
2:  $B_J = \emptyset$ 
3:  $I_{up}(\alpha) \equiv \{t \mid y_t = 1, \alpha_t < C \text{ or } y_t = -1, \alpha_t > 0\}$ .
4: for  $k = 0, 1, 2, \dots, p$  do
5:    $i_k = \text{Arg max}_t \in I_{up}(\alpha)$ 
6:    $I_{low}(\alpha) \equiv \{t \mid y_t = -1, \alpha_t < C \text{ or } y_t = 1, \alpha_t > 0\}$ .
7:   find  $j_i = \text{Arg min}_t \in I_{low}(\alpha)$ 
8:   if  $J_i \neq \emptyset$  then
9:      $B = B \cup i_k \cup J_i$ 

```

Working set selection $WSS - 2^{nd}Order$

The Most Violating Pair (MVP) unfortunately does not converge for all cases in practice. The selection of the working set is critical to the success of the decomposition technique. Fan et al. [23] improved on the MVP method by using second-order information.

Let us define

$$q_t(\alpha) = -y_t \nabla f(\alpha)_t.$$

From (4.7) and (4.8) follows the inequality

$$q_i(\alpha) \leq q_j(\alpha). \tag{4.9}$$

Eq. (4.8) implies that α is a feasible and optimal solution of (4.7) at a stationary point of (4.7). The largest difference $q_i - q_j$ for the pair that most violates (4.9) is compared with

the accuracy requested of the solution. It is used in the stopping criteria of Algorithm 2.

Let us define the quantities:

$$\begin{aligned} a_{ij} &= K_{ii} + K_{jj} - 2K_{ij} > 0, \\ \widehat{a}_{ij} &= \begin{cases} a_{ij}, & \text{if } a_{ij} > 0, \\ \tau & \end{cases} \\ b_{ij} &= q_i(\boldsymbol{\alpha}) - q_j(\boldsymbol{\alpha}) > 0, \end{aligned}$$

where τ is a small positive number. Then one can select

$$i \in \underset{t \in I_{up}(\boldsymbol{\alpha})}{\text{Arg max}} \{q_t(\boldsymbol{\alpha})\}, \quad (4.10)$$

and

$$j \in \underset{t \in I_{low}(\boldsymbol{\alpha})}{\text{Arg min}} \left\{ \frac{-b_{it}^2}{\widehat{a}_{it}} : q_t(\boldsymbol{\alpha}) < q_i(\boldsymbol{\alpha}) \right\}. \quad (4.11)$$

In Yang et al. [74], another higher-order working set selection was proposed where

$$j \in \underset{t \in I_{low}(\boldsymbol{\alpha})}{\text{Arg min}} \left\{ \frac{-b_{it}^2}{\widehat{a}_{it}} : q_t(\boldsymbol{\alpha}) < q_i(\boldsymbol{\alpha}) \right\}, \quad (4.12)$$

and i is slected as (4.10).

The sequential minimal optimization algorithm (*SMO*) uses a single pair $B = \{i, j\}$. The schemes described below show that multiple p working pairs can be selected.

To obtain p working pairs, the p indices i are chosen from I_{up} and the corresponding j are selected from I_{low} . This is a generalization of the *SMO* algorithm [18, 19] described as *IPSMO-2* algorithm in Wei et al. [27]. The *SMO* algorithm selects a single pair of data points as a working set, while *IPSMO-2* selects larger working sets containing p pairs in general. The rule of how pairs are selected is based on the maximum **KKT** violation,

similar to that of SMO .

Algorithm 8 WSS_{2nd} Working set selection using second order information

```

1:  $p = \text{number of pairs}$ 
2:  $B_J = \emptyset$ 
3:  $I_{up}(\boldsymbol{\alpha}) \equiv \{t \mid y_t = 1, \alpha_t < C \text{ or } y_t = -1, \alpha_t > 0\}$ .
4: for  $k = 0, 1, 2, \dots, p$  do
5:    $i_k = \text{Arg max}_t \in I_{up}(\boldsymbol{\alpha})$ 
6:    $I_{low}(\boldsymbol{\alpha}) \equiv \{t \mid y_t = -1, \alpha_t < C \text{ or } y_t = 1, \alpha_t > 0\}$  ..
7:   find  $j_i = \text{Arg min}_t \in I_{low}(\boldsymbol{\alpha}) \left\{ \frac{-b_{it}^2}{\widehat{a}_{it}} : q_t(\boldsymbol{\alpha}) < q_i(\boldsymbol{\alpha}) \right\}$  using WSS second order.
8:   if  $J_i \neq \emptyset$  then
9:      $B = B \cup i_k \cup J_i$ 

```

Suppose that p pairs working set are chosen from the data as the working set for each decomposition iteration. The data points $i_s \in I_{up}$, $s = 1, \dots, p$ are selected in the same way as in WSS using (4.10).

For each $i \in I_{up}$, select $j_i \in I_{low}$ using (4.11).

$$j_i = \text{Arg min}_{t \in I_{low}(\boldsymbol{\alpha})} \left\{ \frac{-b_{it}^2}{\widehat{a}_{it}} : q_t(\boldsymbol{\alpha}) < q_i(\boldsymbol{\alpha}) \right\}.$$

The stopping criteria is

$$\begin{aligned} & \max \{-y_t \nabla f(\boldsymbol{\alpha}) \mid y_t = 1, \alpha_t < C \text{ or } y_t = -1, \alpha_t > 0\} \\ & - \min \{-y_t \nabla f(\boldsymbol{\alpha}) \mid y_t = -1, \alpha_t < C \text{ or } y_t = 1, \alpha_t > 0\} < \varepsilon, \end{aligned}$$

where ε is a user-defined accuracy value.

Working set selection implementations

Limiting the search space for I_{up} and I_{low}

One of the challenges we had with the use of dual decomposition WSS_{MVP} and WSS_{2nd} is that a particular data set point can be selected multiple times in different iterations of decomposition in the working set, often increasing the SVM training time. If there is a

way to control this, we hope to get a faster training time. We propose the *minmaxlimiter* algorithm with the following changes:

- We made a change to exclude elements of α that no longer change. $\|\alpha_t - \alpha_{t-1}\| > \delta_\alpha$. α_{t-1} is the value of α_t in the previous iteration. Where $\alpha_t \in B$ and δ_α is a user defined threshold.
- We added another parameter *minAlphaOpt*, which is the minimum number of times a data index is used in working set. After this threshold, if the $\alpha_{i,i=1,\dots,m}$ value is 0 or C , then that data point index is no longer considered
- We also introduce a parameter *maxAlphaOpt*, which is the maximum number of times we want a data set in our decomposition rounds.

minAlphaOpt = ∞ and *maxAlphaOpt* = ∞ will result in considering every α in every iteration.

Algorithm 9 *minmaxlimiter*

```

1:  $p$  = number of pairs
2:  $B_J = \emptyset$ 
3: if  $(\alpha_t - \alpha_{t-1} > \delta_\alpha)$  &  $(IterCount(\alpha_t) < maxAlphaOpt)$  then
4:   if  $(IterCount(\alpha_t) < minAlphaOpt)$  then
5:      $I_{up}(\alpha) \equiv \{t \mid y_t = 1, \alpha_t < C \text{ or } y_t = -1, \alpha_t > 0\}$ .
6:      $I_{low}(\alpha) \equiv \{t \mid y_t = -1, \alpha_t < C \text{ or } y_t = 1, \alpha_t > 0\}$  ..
7:      $\alpha_{t-1}$  is the previous iteration of  $\alpha$ .
```

Working set selection *minAlphaCheck*

WSS_{MVP} is easier to implement and runs faster than WSS_{2nd} , which requires calculation of the kernels for the evaluation of (4.11). On the good side, WSS_{2nd} is guaranteed to converge. A modification to WSS_{2nd} is the introduction of parameter we will call *minAlphaCheck*. This is used to reduce the search space of j in (4.11) to the first *minAlphaCheck* in the sorted I_{low} set. The experimental results show that this converges and is faster than WSS_{2nd} .

Algorithm 10 $WSS_{minAlphaCheck}$

```
1:  $p$  = number of pairs
2:  $B_J = \emptyset$ 
3:  $I_{up}(\alpha) \equiv \{t \mid y_t = 1, \alpha_t < C \text{ or } y_t = -1, \alpha_t > 0\}$ .
4: for  $k = 0, 1, 2, \dots, p$  do
5:    $i_k = \text{Arg max}_t \in I_{up}(\alpha)$ 
6:    $I_{low}(\alpha) \equiv \{t \mid y_t = -1, \alpha_t < C \text{ or } y_t = 1, \alpha_t > 0\}$ ..
7:    $I_{low}^* \subset I_{low}$ ,  $\alpha$  in top  $minAlphaCheck$  in sorted  $-y_t \nabla f(\alpha)_t$ 
8:   find  $j_{i_k} = \text{Arg min}_t \in I_{low}^*(\alpha)$ ,  $\left\{ \frac{-b_{it}^2}{a_{it}} : q_t(\alpha) < q_i(\alpha) \right\}$  using WSS second order.
9:   if  $j_{i_k} \neq \emptyset$  then
10:     $B = B \cup i_k \cup j_{i_k}$ 
```

Working set selection ($WSS_{minCount}$)

$WSS_{minCount}$ uses a different heuristic method to determine the working set. The method seeks to use all data points first before selecting any previously selected data points are reused. At the beginning, a counter for all α is set to zero; whenever an $\alpha_{i,i=1,\dots,m}$ is used in the working set, the counter is incremented by one. A global counter $minCount$ is also started at 0. This counter is only incremented after all the alphas have been used in the decomposition rounds or there are no more possibilities of adding an alpha to the B . The $WSS_{minCount}$ rule is described as Algorithm 11.

Algorithm 11 $WSS_{minCount}$

```
1:  $p$  = number of pairs
2:  $B_J = \emptyset, I_{Up} = \emptyset, I_{low} = \emptyset$ 
3: while  $I_{Up} = \emptyset, I_{low} = \emptyset$  do
4:   if  $(\alpha_t - \alpha_{t-1} > \delta_\alpha)$  &  $(IterCount(\alpha_t) < maxAlphaOpt)$  then
5:     if  $(IterCount(\alpha_t) < minCount)$  then
6:        $I_{up}(\alpha) \equiv \{t \mid y_t = 1, \alpha_t < C \text{ or } y_t = -1, \alpha_t > 0\}$ ,
        $I_{low}(\alpha) \equiv \{t \mid y_t = -1, \alpha_t < C \text{ or } y_t = 1, \alpha_t > 0\}$ .
7:        $I_{low}^* \subset I_{low}$ ,  $\alpha$  in top  $minAlphaCheck$  in sorted  $-y_t \nabla f(\alpha)_t$ 
8:       if  $I_{Up} = \emptyset, I_{low} = \emptyset$  then
9:          $minCount = minCount + 1$ 
```

Combining all the WSS algorithms discussed in Algorithms 7 - 11, we have the combinations shown in Table 4.1. WSS_0 and WSS_4 are the same as described in [27]. Our contribution to the selection of the working set are the proposed methods WSS_1 , WSS_2 , WSS_3 , WSS_5 , WSS_6 , WSS_7 and WSS_8 methods. We will evaluate the performance of each method with numerical examples.

WSS	WSS Type	Limit type used	Parameters
WSS_0	WSS_{MVP}		
WSS_1	WSS_{MVP}	$WSS_{minmaxlimiter}$	$minAlphaOpt, maxAlphaOpt = \infty$
WSS_2	WSS_{MVP}	$WSS_{minmaxlimiter}$	$minAlphaOpt, maxAlphaOpt$
WSS_3	WSS_{MVP}	$WSS_{minCount}$	$minCount, minAlphaOpt, maxAlphaOpt$
WSS_4	WSS_{2nd}		
WSS_5	WSS_{2nd}	$WSS_{minAlphaCheck}, WSS_{minmaxlimiter}$	$minAlphaCheck$
WSS_6	WSS_{2nd}	$WSS_{minAlphaCheck}, WSS_{minmaxlimiter}$	$minAlphaCheck, minAlphaOpt, maxAlphaOpt = \infty$
WSS_7	WSS_{2nd}	$WSS_{minAlphaCheck}, WSS_{minmaxlimiter}$	$minAlphaCheck, minAlphaOpt, maxAlphaOpt$
WSS_8	WSS_{2nd}	$WSS_{minAlphaCheck}, WSS_{minCount}$	$minAlphaCheck, minCount, minAlphaOpt, maxAlphaOpt$

Table 4.1: WSS combination used in analysis

Summary

In this chapter, we have described decomposition techniques for solving SVM problems and some working set selection methods in the literature. We have proposed new methods for selecting a working set where a multiple number of working set pairs $p > 2$ are selected for each decomposition problem. The numerical results comparing the working set selection schemes are provided in Chapter 6.

Chapter 5: Implementation of Lagrange multiplier SVM solvers

To efficiently solve large-scale SVM problems, there are two important parts:

- The selection of the working set B .
- Solving the subsequent decomposition problem.

The time to determine a working set is in many cases longer than the time to find the solution for the SVM subproblem for that working set (see Fig. 5.1 and 5.2). Hsu and Lin [77] have also shown that if C is large and the Hessian matrix Q is not well-conditioned, the decomposition methods converge very slowly. Thus, it is important to use an efficient working set selection scheme while trying to train the SVM.

Recently, trends in computer architecture have been moving towards increasingly parallel hardware [78]. Most CPUs feature multiple cores, and general purpose graphics processing units (GPUs) that can execute thousands of parallel threads on their hundreds of throughput-optimized cores. Both parallel frameworks offer enormous raw power and have the potential to provide huge speedups; however, to utilize each type of parallel thread effectively, algorithms must be carefully decomposed and optimized in fundamentally different ways [79]. GPUs are based on a Same Instruction Multiple Data (SIMD) architecture, this requires that all threads within one block execute the exact same instructions, whereas multi-core CPUs have much fewer threads with no such restriction. A CPU core on the other hand is designed for very complex control logic that optimizes the execution of sequential programs. A GPU core which is relatively light weight is optimized for data-parallel tasks with simpler control logic, focusing on the throughput of parallel programs [80].

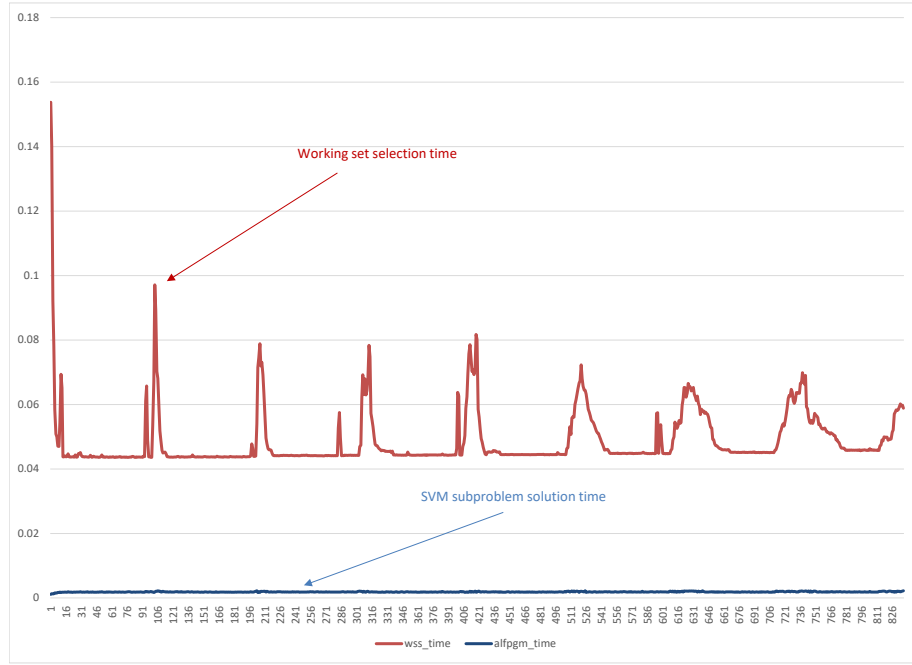


Figure 5.1: Comparison between WSS time and ALFPGM time for webdata_wXa WSS_0

On a high level, there are two different approaches to parallelizing algorithms; explicit and implicit approaches [79]. In the explicit approach, an algorithm is parallelized by finding the independent components of the algorithm which can be run in parallel and encodes this parallelism using some appropriate explicitly parallel language or library such as OpenMP (for multicores), MPI (for clusters), CUDA or OpenCL (for GPUs). In the implicit approach, the algorithm is expressed as a series of operations that are known to be highly parallel and for which highly optimized parallel libraries already exist for most platforms. Examples include libraries for dense linear algebra operations such as PLASMA [81] and Intels MKL [82] for multicores; MAGMA [81], and CuBLAS [83] for GPUs and PDE solvers such as PETSc [84].

For this work, we have used high performance computing techniques to speed up the

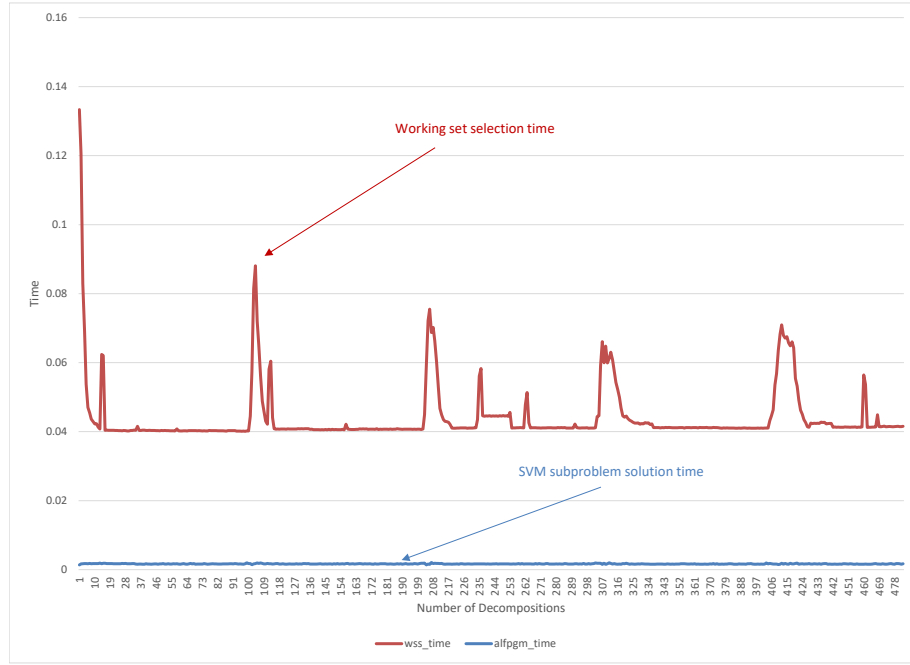


Figure 5.2: Comparison between WSS time and ALFPGM time for webdata_wXa WSS_4

two aspects of getting the SVM solution; the selection of the working set B and solving the subsequent decomposition problem. We looked at using parallel multiprocessing to speed up areas of the code that can be parallelized explicitly. The Intel oneAPI Math Kernel Library (MKL) was utilized for it's high performance math routines. It has BLAS (Basic Linear Algebra Subprograms) optimized for Intel processors. Additionally, we looked at the use of graphical processing units GPU to speed up the algorithm.

To select the working set, we have used the working set selection schemes described in Table 4.1 using Algorithms 7 - 11. The kernel needs to be computed in all cases. The next section describes how we computed the kernel matrix.

Kernel matrix computation

The cache technique is used to store previously computed rows to access them when the same kernel value needs to be recomputed. The amount of cached kernel is a selectable parameter. We have used the same **Least Recently Used** (LRU) LRU strategy that LibSVM uses, and it is a good basis for comparing the results obtained using the SVM training methods. Previous discussions [27, 85] have been held on the ineffectiveness of the LRU strategy. However, that is beyond the scope of this work.

To compute the Kernel matrix, Linear operations using the Intel MKL are used. A row of the kernel matrix can be computed efficiently using

$$xx_i = \langle x_i, x_i \rangle$$

$$K_j = X \times x_j$$

$$K_i = \exp(xx_i + xx_j - 2K_j), j = 1 \cdots n$$

Intel MKL's `cblas_dgemv` is used to compute $y := \alpha * A' * x + \beta * y$.

```
1 void getK_i(svm_problem *alfpgm_prob, double *K_i, int i) {
2     double *Ktemp;
3     int j;
4     int cached_length = get_data(alfpgm_prob, i, &Ktemp);
5     if (cached_length < alfpgm_prob->nData) {
6
7         if (alfpgm_param.use_cache_type == 1) {
8             for (j = 0; j < alfpgm_prob->nData; j++) {
9                 Ktemp[j] = alfpgm_prob->K[get_tri_index(i, j)];
10            }
11        } else {
12            double *Xi;
13            Xi = &alfpgm_prob->X[i * svm_param.nFeatures];
14
15            double *K = (double*) (calloc(alfpgm_prob->nData, sizeof(double)));
16            double xdot = alfpgm_prob->Xdot[i];
17            cblas_dgemv(CblasRowMajor, CblasNoTrans, alfpgm_prob->nData,
18                      svm_param.nFeatures, 1.0, alfpgm_prob->X,
19                      svm_param.nFeatures, Xi, 1, 0, K, 1);
```

```

20 #pragma omp parallel for default(none) shared(xdot, svm_param, Ktemp, K, alfpgm_prob) private(j)
    schedule(guided)
21     for (int j = 0; j < alfpgm_prob->nData; j++) {
22         Ktemp[j] = exp(
23             -svm_param.gamma
24             * (xdot + alfpgm_prob->Xdot[j] - 2 * K[j]));
25     }
26     free(K);
27 }
28 }
29 memcpy(K_i, Ktemp, sizeof(double) * alfpgm_prob->nData);
30 }

```

Listing 5.1: Kernel row computation

OpenMP **parallel for** is used to compute kernel values.

```

1 void getK_i(svm_problem *alfpgm_prob, double *K_i, int i) {
2     double *Ktemp;
3     int j;
4     int cached_length = get_data(alfpgm_prob, i, &Ktemp);
5     if (cached_length < alfpgm_prob->nData) {
6
7         if (alfpgm_param.use_cache_type == 1) {
8             for (j = 0; j < alfpgm_prob->nData; j++) {
9                 Ktemp[j] = alfpgm_prob->K[get_tri_index(i, j)];
10            }
11        } else {
12            double *Xi;
13            Xi = &alfpgm_prob->X[i * svm_param.nFeatures];
14
15            double *K = (double*) (calloc(alfpgm_prob->nData, sizeof(double)));
16            double xdot = alfpgm_prob->Xdot[i];
17            cblas_dgemv(CblasRowMajor, CblasNoTrans, alfpgm_prob->nData,
18                svm_param.nFeatures, 1.0, alfpgm_prob->X,
19                svm_param.nFeatures, Xi, 1, 0, K, 1);
20 #pragma omp parallel for default(none) shared(xdot, svm_param, Ktemp, K, alfpgm_prob) private(j)
    schedule(guided)
21     for (int j = 0; j < alfpgm_prob->nData; j++) {
22         Ktemp[j] = exp(
23             -svm_param.gamma
24             * (xdot + alfpgm_prob->Xdot[j] - 2 * K[j]));
25     }
26     free(K);
27 }
28 }

```

```

29 memcpy(K_i, Ktemp, sizeof(double) * alfpgm_prob->nData);
30 }

```

Listing 5.2: Individual kernel matrix value computation

Gradient ∇f update

Selection of the working set requires the calculation of the gradient ∇f .

$$\nabla f(\alpha, \lambda) = Q\alpha + (q - e) - (\lambda - \mu \sum_{i=1}^m y_i \alpha_i) y$$

Using the method described in [22], the delta changes in α_B , $\Delta\alpha_B$ after each decomposition is computed, and this is used to update $\nabla f(\alpha)_i$. To further simplify our solution, at the start of the algorithm, $\nabla f(\alpha)_i = y_i$. The gradient is updated after each decomposition.

$$\nabla f(\alpha)_i = (\Delta\alpha_B Q_{BB} + \Delta\alpha_B Q_{BN}) y_B \quad (5.1)$$

```

1 void updateF(double *minusF, svm_problem *alfpgm_prob, const double *H_B_N,
2             double *alpha, const double *alpha_differences_vector) {
3     double max_val = -DBL_MAX;
4     double min_val = DBL_MAX;
5     int i, idx;
6     double *alphaHVec = (double*) (calloc(alfpgm_prob->nData, sizeof(double)));
7     cblas_dgemv(CblasRowMajor, CblasTrans, alfp_gm_param.nWss,
8                alfp_gm_prob->nData, 1, H_B_N, alfp_gm_prob->nData,
9                alpha_differences_vector, 1, 0, alphaHVec, 1);
10    {
11        for (i = 0; i < alfp_gm_prob->nData; i++) {
12            idx = alfp_gm_prob->wss_index[i];
13            minusF[idx] -= alfp_gm_prob->y[idx] * alphaHVec[i];
14        }
15    }
16    for (i = 0; i < alfp_gm_prob->nData; i++) {
17        if ((alfpgm_prob->active[i])) {
18            if (alfpgm_prob->y[i] > 0) {
19                if (!is_upper_bound_alpha(alpha[i])) {
20                    max_val = fmax(max_val, minusF[i]);
21                }

```

```

22     if (!is_lower_bound_alpha(alpha[i])) {
23         min_val = fmin(min_val, minusF[i]);
24     }
25 }
26 if (alfpgm_prob->y[i] < 0) {
27     if (!is_lower_bound_alpha(alpha[i])) {
28         max_val = fmax(max_val, minusF[i]);
29     }
30     if (!is_upper_bound_alpha(alpha[i])) {
31         min_val = fmin(min_val, minusF[i]);
32     }
33 }
34 }
35 }
36 free(alphaHVec);
37 alfpgm_prob->gap = (max_val - min_val);
38 }

```

Listing 5.3: Updating the gradient of $\nabla f(\alpha)$

GPU implementation

A challenge with GPU is the movement of data between the host system and the device (GPU). GPUs have a smaller memory in comparison to the host system. Our analysis shows that most of the time is spent in the selection of the working set described below. This makes the GPU inappropriate for constant matrix computation.

Augmented Lagrangian - Fast Projected Gradient Method for SVM

Let $\{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_m, y_m)\}$ be a set of m labeled data points where $\mathbf{x}_i \in \mathbb{R}^n$ is a vector of features and $y_i \in \{-1, 1\}$ represent the label that indicates the class to which \mathbf{x}_i belongs. To train the SVM, we need to find $\alpha^* = (\alpha_1^*, \dots, \alpha_m^*)^T$ that solves the following problem:

$$\begin{aligned}
\min_{\alpha} \quad & f(\alpha) = -\sum_{i=1}^m \alpha_i + \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j \mathbf{K}(\mathbf{x}_i, \mathbf{x}_j) \\
\text{s.t.} \quad & \sum_{i=1}^m \alpha_i y_i = 0, \\
& 0 \leq \alpha_i \leq C, i = 1, \dots, m.
\end{aligned} \tag{5.2}$$

The matrix Q with elements $Q_{ij} = y_i y_j K(x_i, x_j)$ is positive semidefinite but usually dense. Therefore, a large number of training points m result in significant operating memory requirements, leading to inefficiencies for many general purpose optimization algorithms.

A relatively simple and efficient method, ALFPGM, that is capable of training medium-size SVMs with up to a few tens of thousands of data points was proposed in [86]. The algorithm takes advantage of two methods:

- an augmented Lagrangian method [62, 63] used to satisfy the only linear equality constraint and
- a fast projected gradient method or FISTA (Algorithms 1-5) to solve a minimization problem on box constraints [45, 48, 67]

The ALFPGM projects the dual variables onto the “box-like” set: $0 \leq \alpha_i \leq C, \quad i = 1, \dots, m$.

Using the following definitions

$$\begin{aligned}
f(\alpha) &= \sum_{i=1}^m \alpha_i (q_i - 1) + \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j \mathbf{K}(\mathbf{x}_i, \mathbf{x}_j) \\
g(\alpha) &= \sum_{i=1}^m y_i \alpha_i + G_k
\end{aligned} \tag{5.3}$$

and the bounded set:

$$Box = \{\boldsymbol{\alpha} \in \mathbb{R}^m : 0 \leq \alpha_i \leq C, i = 1, \dots, m\}, \quad (5.4)$$

the optimization problem (5.2) can be rewritten as follows:

$$\begin{aligned} \min_{\boldsymbol{\alpha}} \quad & f(\boldsymbol{\alpha}) \\ \text{s.t.} \quad & g(\boldsymbol{\alpha}) = 0, \\ & \boldsymbol{\alpha} \in Box \end{aligned} \quad (5.5)$$

The augmented Lagrangian is defined as

$$\mathcal{L}_\mu(\boldsymbol{\alpha}, \lambda) = f(\boldsymbol{\alpha}) - \lambda g(\boldsymbol{\alpha}) + \frac{\mu g(\boldsymbol{\alpha})^2}{2}, \quad (5.6)$$

where $\lambda \in \mathbb{R}$ the Lagrange multiplier that corresponds to the only equality constraint and $\mu > 0$ is the scaling parameter.

The augmented Lagrangian method is a sequence of inexact minimizations of $\mathcal{L}_\mu(\boldsymbol{\alpha}, \lambda)$ in $\boldsymbol{\alpha}$ on the *Box* set

$$\hat{\boldsymbol{\alpha}} \approx \boldsymbol{\alpha}(\lambda) = \underset{\boldsymbol{\alpha} \in Box}{\operatorname{argmin}} \mathcal{L}_\mu(\boldsymbol{\alpha}, \lambda) \quad (5.7)$$

followed by updating the Lagrange multiplier λ :

$$\hat{\lambda} = \lambda - \mu g(\hat{\boldsymbol{\alpha}}). \quad (5.8)$$

For the stopping criteria for (5.7), we use the following function that measures the violation of the first order optimality conditions of (5.7):

$$\Omega(\boldsymbol{\alpha}, \lambda) = \max_{1 \leq i \leq m} \Omega_i(\boldsymbol{\alpha}, \lambda), \quad (5.9)$$

where

$$\Omega_i(\boldsymbol{\alpha}, \lambda) = \begin{cases} |\nabla_{\boldsymbol{\alpha}} \mathcal{L}_{\mu}(\boldsymbol{\alpha}, \lambda)_i| & 0 < \alpha_i < C, \\ \max \{0, -\nabla_{\boldsymbol{\alpha}} \mathcal{L}_{\mu}(\boldsymbol{\alpha}, \lambda)_i\} & \alpha_i = 0, \\ \max \{0, \nabla_{\boldsymbol{\alpha}} \mathcal{L}_{\mu}(\boldsymbol{\alpha}, \lambda)_i\} & \alpha_i = C. \end{cases} \quad (5.10)$$

The inner FPGM(FISTA) loop is terminated when $\min(\|g(\boldsymbol{\alpha})\|, \Omega(\boldsymbol{\alpha}, \lambda)) < \epsilon$

For the final stopping criteria for the augmented Lagrangian method, we use $\mu(\boldsymbol{\alpha}, \lambda) = \max \{\Omega(\boldsymbol{\alpha}, \lambda), |g(\boldsymbol{\alpha})|\}$, which measures the violation of the optimality conditions for (5.10).

Algorithm 12 Augmented Lagrangian-Fast Projected Gradient Method - No decomposition case

Initialization

- 1: $\boldsymbol{\alpha}^{[0]} \in \mathbb{R}^m$ (starting point, not necessarily feasible) $\boldsymbol{\alpha}^{[0]} = \mathbf{0}$
- 2: $\lambda^{[0]} \in \mathbb{R}$ (initial "guestimate" of Lagrange multiplier vector) $\lambda^{[0]} = 0$
- 3: $\mu_0 > 0$ (initial value of scaling parameter)
- 4: $\epsilon > 0$ (Required accuracy)
- 5: $0 < \theta < 1$
- 6: $\delta \geq 1$
- 7: $rec = \max \{\Omega(\boldsymbol{\alpha}, \lambda), |g(\boldsymbol{\alpha})|\}$

Run the Augmented Lagrangian Algorithm

- 1: **while** $rec > \epsilon$ **do**
- 2: find $\boldsymbol{\alpha} : \Omega(\boldsymbol{\alpha}, \lambda) \leq \theta * rec$ using FPGM
- 3: $\lambda = \lambda - \mu g(\boldsymbol{\alpha})$
- 4: $rec = \min(rec, \Omega(\boldsymbol{\alpha}, \lambda))$
- 5: $k = \delta k$

Run the FPGM

- 1: **procedure** FPGM
- 2: Input $(\boldsymbol{\alpha}, \lambda)$
- 3: Set $\bar{\boldsymbol{\alpha}} = \boldsymbol{\alpha}, t = 1$ Select $L > 0$
- 4: **while** $\Omega(\boldsymbol{\alpha}, \lambda) > \theta * rec$ **do**
- 5: Set $\hat{\boldsymbol{\alpha}} = P_{Box}(\boldsymbol{\alpha} - \frac{1}{L} \nabla_{\boldsymbol{\alpha}} \mathcal{L}_{\mu}(\boldsymbol{\alpha}, \lambda))$.
- 6: Set $\bar{t} = 0.5(1 + \sqrt{1 + 4t^2})$.
- 7: Set $\boldsymbol{\alpha} = \hat{\boldsymbol{\alpha}} + \frac{t-1}{\bar{t}}(\hat{\boldsymbol{\alpha}} - \bar{\boldsymbol{\alpha}})$.
- 8: $\bar{\boldsymbol{\alpha}} = \hat{\boldsymbol{\alpha}}, t = \bar{t}$.
- 9: Output $\hat{\boldsymbol{\alpha}}$.

- 1: **procedure** P_{Box}
 - 2: **for** $i = 1, \dots, m$ **do**
 - 3: **if** $\alpha_i < 0$ **then**
 - 4: $\alpha_i = 0$
 - 5: **if** $\alpha_i > C$ **then**
 - 6: $\alpha_i = C$
-

The fast projected gradient method (FPGM) requires estimation of the Lipschitz constant $L > 0$ of the gradient $\nabla_{\alpha}\mathcal{L}_{\mu}$ so that the inequality.

$$\|\nabla_{\alpha}\mathcal{L}_{\mu}(\alpha_1, \lambda) - \nabla_{\alpha}\mathcal{L}_{\mu}(\alpha_2, \lambda)\| \leq L \|\alpha_1 - \alpha_2\| \quad (5.11)$$

holds for any $\alpha_1, \alpha_2 \in \mathbb{R}^m$ [14, 45, 48]. The gradient and the Hessian of $\mathcal{L}_{\mu}(\alpha, \lambda)$ are as follows:

$$\begin{aligned} \nabla_{\alpha}\mathcal{L}_{\mu}(\alpha, \lambda) &= Q\alpha + (q - e) - (\lambda - \mu \sum_{i=1}^m y_i \alpha_i)y, \\ \nabla^2_{\alpha\alpha}\mathcal{L}_{\mu}(\alpha, \lambda) &= Q + \mu yy^T. \end{aligned} \quad (5.12)$$

where Q is an m by m matrix with the elements $Q_{ij} = y_i y_j K(x_i, x_j)$, $i = 1, \dots, m$, $j = 1, \dots, m$, $y = (y_1, \dots, y_m)^T$, $e = (1, \dots, 1)^T$.

Since \mathcal{L}_{μ} is of quadratic form with respect to α ,

$$\mathcal{L}_{\mu} = \|Q + \mu yy^T\| \leq \|Q\| + m\mu \quad (5.13)$$

where the matrix spectral norm is the largest singular value of a matrix.

Algorithm 5 describes the ALFPGM. This algorithm, as described, does not use any decomposition or working set selection technique. It uses all training data points and serves as a reference for comparison with a more advanced algorithm that takes advantage of working sets. The convergence of Algorithm 5 is established in [49].

Working set selection for ALFPGM

We extend the above algorithm by using the decomposition algorithms described in Table 4.1 using Algorithms 7 - 11 The SVM problem becomes

$$\begin{aligned}
\min_{\alpha_B} \quad & \frac{1}{2} \alpha_B^T \mathbf{Q}_{BB} \alpha_B + \alpha_B^T q_{BN} \\
\text{s.t.} \quad & \alpha_B^T y_B + g_k = 0, \\
& 0 \leq \alpha_B \leq C.
\end{aligned} \tag{5.14}$$

where $q_{BN} = \mathbf{Q}_{BN} \alpha_N - \mathbf{e}_B$ and $g_k = \alpha_N^T y_N$

Algorithm 13 SVM ALFPGM Decomposition

- 1: $\nabla f(\alpha) = y$
 - 2: **while** global minimum not reached **do**
 - 3: find working set B
 - 4: find the Kernel submatrix $Q = Q_{BB} + Q_{BN}$
 - 5: find α_B for working set B using the ALFPGM.
 - 6: $\nabla f(\alpha)_i = (\Delta \alpha_B Q_{BB} + \Delta \alpha_B Q_{BN}) y_B$
 - 7: Update α with α_B
-

Implementation of ALFPGM

CPU implementation

The ALFPGM, being a first-order method, uses vector-vector and matrix-vector multiplications. This makes it ideal for use in parallel computing environments using linear algebra libraries such as BLAS numerical libraries.

The ALFPGM implemented and tested on a Linux workstation with two Intel[®] Xeon[®] 2.50GHz each with 12 cores, 64 GB RAM. Each processor has its own memory interface and is connected to half of the installed RAM, totaling 64GB.

FISTA procedure

For the inner FPGM procedure, we evaluate the FISTA algorithms described in Algorithms 1-5. They are suitable to be vectorized and parallelized. The updates and the projector operations are well suited for parallelizing using multiple cores or a GPU.

The operations below were implemented using OpenMP parallel for.

$$\begin{aligned} x_k &:= p_L(y_k), \\ t_{k+1} &:= 1 + \frac{\sqrt{1+4t_k^2}}{2}, \\ y_{k+1} &:= x_k + \left(\frac{t_k-1}{t_{k+1}}\right)(x_k - x_{k-1}). \end{aligned}$$

Computing the Lipschitz constant L

The fast projected gradient method (FPGM) requires estimation of the Lipschitz constant L .

Let the symmetric $m \times m$ matrix Q have an eigenvalue, λ_1 , the largest eigenvalue of a magnitude much larger than the remaining eigenvalues. The largest eigenvalue can be efficiently computed using the power method. Let λ_1 be the dominant eigenvalue of A and $v = c_1x_1 + \dots + c_nx_n$ an arbitrary vector such that $c_1 \neq 0$. Then

$$\begin{aligned} Q^k v &= c_1 \lambda_1^k x_1 + \dots + c_n \lambda_n^k x_n \\ &= \lambda_1^k [c_1 x_1 + c_2 (\lambda_2/\lambda_1)^k x_2 + \dots + c_n (\lambda_n/\lambda_1)^k x_n] \end{aligned} \tag{5.15}$$

$$\text{with } v^{(k)} = Q^k v / \lambda_1^k = c_1 x_1 + \underbrace{c_2 (\lambda_2/\lambda_1)^k x_2 + \dots + c_n (\lambda_n/\lambda_1)^k x_n}_{\Delta_k}.$$

The vector $v^{(k)}$ converges to $c_1 x_1$. Moreover, $\|\Delta_k\| = \|v^{(k)} - c_1 x_1\| \leq \text{const} \cdot r^k$ where $r = |\lambda_2/\lambda_1| < 1$.

Therefore, the vectors $Q^k v$ (obtained by the powers of A) will align in the direction of the dominant eigenvector x_1 as $k \rightarrow \infty$. The number r characterizes the speed of alignment,

i.e. the speed of convergence $\|\Delta_k\| \rightarrow 0$. Note that if $c_2 \neq 0$, then

$$\|v^{(k+1)} - c_1 x_1\| / \|v^{(k)} - c_1 x_1\| \rightarrow r.$$

The number r is called the *convergence ratio* or the *contraction number*. The largest eigenvalue of Q does not need to be accurate, a good estimate is sufficient. So, from (5.13) above, the Lipschitz constant L does not exceed $\hat{\lambda}_1 + m\mu$ i.e.

$$L \leq \hat{\lambda}_1 + m\mu \quad (5.16)$$

where $\hat{\lambda}_1$ is the maximum eigenvalue of Q obtained using the power method.

```

1 double computeL(svm_problem *alfpgm_prob, double *HBB) {
2     int i, j;
3     double eig_max = -DBL_MAX, ldiff = DBL_MAX, zprevious;
4     double Hx;
5     double *z_eig, *x_eig;
6     z_eig = (double*) (calloc(alfpgm_param.nWss, sizeof(double)));
7     x_eig = (double*) (calloc(alfpgm_param.nWss, sizeof(double)));
8     for (j = 0; j < alfpgm_param.nWss; j++) {
9         x_eig[j] = (double) (rand()) / (double) ((RAND_MAX));
10    }
11    while (ldiff > .01) {
12        zprevious = eig_max;
13
14        eig_max = 0;
15
16    #pragma omp parallel default(none) shared(HBB, alfpgm_param, z_eig, x_eig) reduction(+ : Hx)
17        reduction(max : eig_max)
18    {
19    #pragma omp for private(i, j) schedule(static)
20        for (i = 0; i < alfpgm_param.nWss; i++) {
21            Hx = 0;
22            for (j = 0; j < alfpgm_param.nWss; j++) {
23                Hx += (HBB[i * alfpgm_param.nWss + j]) * x_eig[j];
24            }
25            z_eig[i] = Hx;
26            eig_max = fmax(eig_max, fabs(Hx));
27        }
28    }
29    }

```

```

28
29     for (i = 0; i < alfpgm_param.nWss; i++) {
30         x_eig[i] = z_eig[i] / eig_max;
31     }
32     ldiff = fabs(eig_max - zprevious);
33 }
34 free(z_eig);
35 free(x_eig);
36 return (eig_max);
37 }

```

Listing 5.4: Computing the Lipschitz constant

NRAL for SVM

To solve the SVM problem, we need to find the optimal variables $\boldsymbol{\alpha}^* = (\alpha_1^*, \dots, \alpha_m^*)^T$ that solve the following decomposition problem:

$$\begin{aligned}
 \min_{\boldsymbol{\alpha}_B} \quad & \frac{1}{2} \boldsymbol{\alpha}_B^T \mathbf{Q}_{BB} \boldsymbol{\alpha}_B + \boldsymbol{\alpha}_B^T q_{BN} \\
 \text{s.t.} \quad & \boldsymbol{\alpha}_B^T y_B + g_k = 0,
 \end{aligned} \tag{5.17}$$

$$0 \leq \boldsymbol{\alpha}_B \leq C.$$

where $q_{BN} = \mathbf{Q}_{BN} \boldsymbol{\alpha}_N - \mathbf{e}_B$ and $g_k = \boldsymbol{\alpha}_N^T y_N$

The NRAL technique solves the problem expressed in (5.17) by constructing an equivalent problem in which the constraints are rescaled by a function $\psi : -\infty \leq t_0 < t < t_1 \leq +\infty$ such that $\psi(t)$ has the following properties:

$$\left. \begin{aligned}
 \psi(0) &= 0, \quad \psi'(t) > 0, \quad \psi'(0) = 1, \quad \psi''(t) < 0 \\
 \psi'(t) &\leq \frac{a}{(t+1)} \\
 -\psi''(t) &\leq \frac{b}{(t+1)^2}
 \end{aligned} \right\} \quad t \geq 0, a > 0, b > 0$$

For $\mu > 0$, the following problem is equivalent to the original problem:

$$\begin{aligned}
\min_{\alpha_B} \quad & \frac{1}{2} \alpha_B^T \mathbf{Q}_{BB} \alpha_B + \alpha_B^T q_{BN} \\
\text{s.t.} \quad & \alpha_B^T y_B + g_\mu = 0, \\
& \frac{1}{\mu} \psi(\mu \alpha_B(\mathbf{x})) \geq 0, \quad j = 1, \dots, p.
\end{aligned} \tag{5.18}$$

NRAL converges for any fixed scaling parameter $\mu > 0$ due to the Lagrange multipliers update [12].

In [73] the transformation ψ was chosen as

$$\psi(t) = \begin{cases} \log(t+1) & t > -0.5 \\ -2t^2 + \log(.5) + .5 & t \leq -0.5 \end{cases}$$

The Augmented Lagrangian for this equivalent problem is

$$\mathcal{L}_\mu(\alpha, \lambda, \nu) = f(\alpha) - \lambda^T g(\alpha) - \frac{1}{\mu} \sum_{i=1}^p \nu_i \psi(\mu c_i(\alpha)) + \frac{\mu}{2} \sum_{j=1}^q (g_j(\alpha))^2$$

where $\alpha \in \mathbb{R}^p$ and $\lambda \in \mathbb{R}^q$ are dual variables.

The NRAL technique finds the next iterate $(\alpha^{s+1}, \lambda^{s+1}, \nu^{s+1})$ is found by solving the unconstrained minimization problem

$$\alpha^{s+1} = \arg \min_{\alpha \in \mathbb{R}^n} \mathcal{L}_\mu(\alpha^s, \lambda^s, \nu^s)$$

where $(\alpha^s, \lambda^s, \nu^s)$ is a current iterate and updating the multipliers

$$\begin{aligned}\lambda_i^{s+1} &= \lambda_i^s \psi'(\mu c_i(x^{s+1})), \quad i = 1, \dots, p. \\ \nu_j^{s+1} &= \nu_j^s - \mu g_j(x^{s+1}), \quad j = 1, \dots, q.\end{aligned}$$

At each iteration of the NRAL technique, any unconstrained minimization routine can be used to find $\arg \min_{\alpha \in \mathbb{R}^n} \mathcal{L}_\mu(\alpha^s, \lambda^s, \nu^s)$. If $(\alpha_*, \lambda_*, \nu_*)$ is the solution, then for μ large enough, under second-order optimality conditions, the iterates $(\alpha^s, \lambda^s, \nu^s)$ derived using the NRAL technique will converge to the solution linearly.

The gradient and the Hessian of $\mathcal{L}_\mu(\alpha, \lambda, \nu)$ are as follows

$$\begin{aligned}\nabla_\alpha \mathcal{L}_\nu(\alpha, \lambda, \nu) &= \mathbf{Q}\alpha + (\mathbf{Q}_{\text{BN}}\alpha_N - \mathbf{e}_B) - (\lambda - \nu g(\alpha))\mathbf{y} - \lambda_{L_i}\psi'(\nu\alpha_i) + \lambda_{U_i}\psi'(\nu(C - \alpha_i)) \\ \nabla^2_{\alpha\alpha} \mathcal{L}_\nu(\alpha, \lambda, \nu) &= \mathbf{Q} + \nu\mathbf{y}\mathbf{y}^T - \nu\lambda_{L_i}\psi''(\nu\alpha_i) - \nu\lambda_{U_i}\psi''(\nu(C - \alpha_i))\end{aligned}\tag{5.19}$$

Each iteration uses Newton's method to find the minimum of each NRAL step.

$$\begin{aligned}H &= \nabla^2 \mathcal{L}_\mu(\alpha, \lambda, \nu) \\ G &= \nabla \mathcal{L}_\mu(\alpha, \lambda, \nu) \\ \Delta\alpha &= -H \setminus G. \\ \hat{\alpha} &= \alpha + \Delta\alpha\end{aligned}\tag{5.20}$$

The LAPACK library in Intel's MKL [82] is used to obtain $\Delta\alpha$ in (5.20). The decomposition technique is used for the SVM problem similar to ALFPGM. The selection of working sets described earlier in Algorithms 7 - 11 is used. After each decomposition solution, the gradient update technique described earlier (5.1) is used to update the gradient.

```

1
2 double dotProduct(double *vecA, double *vecB, int n) {
3     double dotP = cblas_ddot(n, vecA, 1, vecB, 1);
4     return (dotP);
5 }
```

Algorithm 14 Nonlinear Rescaling Method

Initialization

- 1: $\alpha^{[0]} \in \mathbb{R}^m$ (starting point, not necessarily feasible) $\alpha^{[0]} = 0$
- 2: $\lambda^{[0]} \in \mathbb{R}$ (initial "guestimate" of Lagrange multiplier vector) $\lambda^{[0]} = 0$
- 3: $\mu_0 > 0$ (initial value of scaling parameter)
- 4: $\epsilon > 0$ (Required accuracy)
- 5: $0 < \theta < 1$

Run the NRAL Algorithm

- 1: **while** $rec > \epsilon$ **do**
- 2: find $\alpha : \Omega(\alpha, \lambda, \nu) \leq \theta * rec$ using Newton
- 3: $\nu_i^{s+1} = \nu_i^s \psi'(\mu c_i(x^{s+1}))$, $i = 1, \dots, p$
- 4: $\lambda_j^{s+1} = \lambda_j^s - \mu g_j(x^{s+1})$, $j = 1, \dots, q$
- 5: $rec = \min(rec, \Omega(\alpha, \lambda))$

Run the Newton

- 1: **procedure** NEWTON
- 2: Input (α, λ, ν)
- 3: Set $\bar{\alpha} = \alpha, t = 1$ Select $L > 0$
- 4: **while** $\Omega(\alpha, \lambda, \nu) > \theta * rec$ **do**
- 5: $H = \nabla^2 \mathcal{L}_\mu(\alpha, \lambda, \nu)$
- 6: $G = \nabla \mathcal{L}_\mu(\alpha, \lambda, \nu)$
- 7: $\Delta \alpha = -H \setminus G$.
- 8: Use Armijo line search to get $\hat{\alpha}$
- 9: Output $\hat{\alpha}$.

Armijo Line Search

- 1: β is the step size reduction factor.
 - 2: Start with $\eta^k = s, \beta < 1, \sigma < 1$
 - 3: If $f(\alpha^k + \eta^k \Delta \alpha^k) + \sigma \eta^k (-\nabla f(\alpha^k)^T \Delta \alpha^k) < f(\alpha^k)$
 - 4: then STOP
 - 5: else $\eta^k \leftarrow \beta \eta^k$ and repeat
-

```

6
7 void compute_hessian(int nWss, double mu, double *H, double *Q, double *y_wss,
8   double *lambda_l, double *alpha_wss_nral, double *lambda_u) {
9
10  //HN = Q + mu*(Aeq_2) - mu * diag(lambda_l.*get_h_psi(mu*alpha) + lambda_u.*get_h_psi(mu*(C -
11    alpha)))));
12  double kappa = nral_param.kappa;
13  double C = svm_param.C;
14  #pragma omp parallel for default(none) shared(Q, H, y_wss, mu, kappa, nWss, lambda_l, lambda_u)
15    private(k, j) schedule(guided)
16
17  for (int k = 0; k < nWss; k++) {
18    for (int j = 0; j < nWss; j++) {
19      H[k * nWss + j] = Q[k * nWss + j] + mu * y_wss[j] * y_wss[k];
20    }
21    H[k * nWss + k] += (kappa
22      * (lambda_l[k] * get_hpsi_val(mu * alpha_wss_nral[k])
23        + lambda_u[k])

```

```

24         * get_hpsi_val(
25             mu * (C - alpha_wss_nral[k])));
26
27     }
28 }

```

Listing 5.5: Computing $\Delta\alpha = -H \setminus G$

Summary

In this chapter, we have described the implementations of the selection methods for the working set described in Chapter 4. We have described the gradient update performed after each decomposition. We have also described the implementation of ALFPGM and NRAL.

Chapter 6: Numerical examples SVM training using WSS selection

Numerical Results

ALFPGM and NRAL were first prototyped in MATLAB to get a sense of the performance of the algorithms. Subsequently, they were implemented in C++ and optimized using specialized linear algebra routines in Intel MKL and parallel programming paradigms in OpenMP. All test results reported in this section were performed on a desktop with dual Intel® Xeon® 2.50GHz, 12 Core processors that share 64 GB of computer memory. We selected the data listed in Tables 6.1 and 6.2 from the University of California, Irvine (UCI) Machine Learning Repository [87] and <https://www.openml.org/>. The data are for binary classification with cases where the classes are balanced and unbalanced.

A learning problem must have a measure that explains how well an algorithm performs the task. A universal goal for any (batch) supervised learning algorithm is generalization, which estimates how well the algorithm will perform on future data [88]. The generalization performance of a supervised learning algorithm is limited by three sources of error:

- The *approximation error* that measures how well the exact solution can be approximated by a function implementable by our learning system,
- The *estimation error* which measures how accurately we can determine the best function implementable by our learning system using a finite training set instead of the unseen testing examples. The estimation error is determined by the number of training examples and the capacity of the family of functions[9].
- The *optimization error* which measures how closely we compute the function that best satisfies whatever information can be exploited in our finite training set.

According to Steinwart [89], well-designed compromises lead to estimation and approximation errors that scale between the inverse and the inverse square root of the number of examples [89]. Optimization algorithms, on the other hand, are mainly focused on designing algorithms whose error decreases exponentially or faster with the number of iterations. The training time for each iteration grows linearly or quadratically with the number of training dataset. According to Bottou et al. [9], it is often desirable to use poorly regarded optimization algorithms that trade asymptotic accuracy for lower iteration complexity. Asymptotic performance, i.e. how quickly the accuracy of the solution increases with computing time, is critical to accessing the performance of any learning algorithm. For this work, we will focus on the classification error in comparing between the SVM training algorithms discussed. The same training data are supplied to all algorithms, i.e. they are solving the same problems. Therefore, the classification error obtained from misclassifying the training data can be used to evaluate the performance of the algorithm.

Comparison of SVM training times using ALFPGM and NRAL-Newton with results obtained using MATLABs *QuadProg*

The first set of experiments to test the performance of the algorithms is to obtain the SVM training times using the Augmented Lagrangian Fast Projected Gradient Method (ALFPGM) and Newton Nonlinear Scaling (NRAL-Newton) were compared with SVM training times obtained using MATLAB's *QuadProg*. The results of numerical experiments with ALFPGM and NRAL-Newton were implemented in MATLAB[©] 2020b. The results obtained were compared with the highly optimized interior-point method (IPM) that comes with MATLAB's *QuadProg*. For both ALFPGM, NRAL-Newton, and the IPM, we compare the efficiency of the SVM training with and without the decomposition method. WSS_4 was used as the decomposition technique.

For testing, we have selected 20 binary classification training data sets (Tests 1-20 in Tables 6.1 and 6.2) with a number of training examples ranging from 100 to 19,020. In all cases, the data were normalized and $C = 100$ was used for all tests. The scaling parameter

μ used for all experiments is $\mu = 0.1$ for ALFPGM and $\mu = 250$ for NRAL-Newton. The accuracy of the solution was selected $\varepsilon = 10^{-5}$. The γ values in Table 6.1 were selected using a **grid search** in γ using the cross-validation technique [22, 90].

Table 6.1: Data used for testing (Tests 1 - 17).

	test	m	n	# -1	# 1	γ	Description
1	Arcene	100	10000	44	56	0.500	ARCENE's task is to distinguish cancer versus normal patterns from mass-spectrometric data. This is a two-class classification problem with continuous input variables. This dataset is one of 5 datasets of the NIPS 2003 feature selection challenge.
2	Balance Scale	625	4	288	337	0.500	Balance scale weight & distance database
3	BanknoteAuthentication	1348	4	610	738	0.500	Data were extracted from images that were taken for the evaluation of an authentication procedure for banknotes.
4	BreastCancerWisconsin	569	30	357	212	0.500	Diagnostic Wisconsin Breast Cancer Database
5	CNAE9DataSet	1052	856	112	940	0.500	This is a data set containing 1080 documents of free text business descriptions of Brazilian companies categorized into a subset of 9 categories
6	ContraceptiveMethodChoice	1390	9	614	776	2.000	Dataset is a subset of the 1987 National Indonesia Contraceptive Prevalence Survey.
7	Dexter	600	20000	300	300	0.500	DEXTER is a text classification problem in a bag-of-word representation. This is a two-class classification problem with sparse continuous input variables. This dataset is one of five datasets of the NIPS 2003 feature selection challenge.
8	dorothea	800	100000	78	722	0.500	DOROTHEA is a drug discovery dataset. Chemical compounds represented by structural molecular features must be classified as active (binding to thrombin) or inactive.
9	eegEyeState	14980	14	8257	6723	0.500	The data set consists of 14 EEG values and a value indicating the eye state.
10	EpilepticSeizureRecognition	11500	178	2300	9200	0.500	This dataset is a pre-processed and re-structured/resized version of a very commonly used dataset featuring epileptic seizure detection.
11	GasSensorArrayDrift	13910	128	5491	8419	0.500	This archive contains 13910 measurements from 16 chemical sensors utilized in simulations for drift compensation in a discrimination task of 6 gases at various levels of concentrations.
12	HabermanSurvival	289	3	210	79	0.500	Dataset contains cases from study conducted on the survival of patients who had undergone surgery for breast cancer
13	HCVEgyptian	1385	29	668	717	0.500	Egyptian patients who underwent treatment dosages for HCV about 18 months. Discretization should be applied based on expert recommendations; there is an attached file shows how.
14	HumanActivityRecognition	4224	561	2163	2061	0.500	Human Activity Recognition database built from the recordings of 30 subjects performing activities of daily living (ADL) while carrying a waist-mounted smartphone with embedded inertial sensors.
15	MadelonDataSet	2600	500	1300	1300	0.500	MADELON is an artificial dataset, which was part of the NIPS 2003 feature selection challenge. This is a two-class classification problem with continuous input variables. The difficulty is that the problem is multivariate and highly non-linear.
16	magictlescope	19020	11	12332	6573	0.500	Data are MC generated to simulate registration of high energy gamma particles in an atmospheric Cherenkov telescope
17	PageBlocksClassification	5473	10	4889	514	2.000	The problem consists of classifying all the blocks of the page layout of a document that has been detected by a segmentation process.

Table 6.2: Data used for testing (Tests 18 - 30).

	test	m	n	# -1	# 1	γ	Description
18	Seeds	210	7	70	140	0.500	Measurements of geometrical properties of kernels belonging to three different varieties of wheat. A soft X-ray technique and GRAINS package were used to construct all seven, real-valued attributes.
19	StatSatLog	4435	36	961	3474	0.500	Multi-spectral values of pixels in 3x3 neighbourhoods in a satellite image, and the classification associated with the central pixel in each neighbourhood. Classification is based on red soil.
20	ZeroPenDigits	7494	16	3715	3779	0.500	Digit database of 250 samples from 44 writers
21	webdata_wXa	36974	123	8874	28100	1.000	https://www.openml.org/d/350
22	2d_planes	26714	10	13369	13345	2.000	https://www.openml.org/d/727
23	bank-marketing	45211	16	5289	39922	2.000	https://www.openml.org/d/1461 The data is related with direct marketing campaigns of a Portuguese banking institution. The marketing campaigns were based on phone calls. Often, more than one contact to the same client was required, in order to access if the product (bank term deposit) would be (or not) subscribed.
24	riccardo	18201	4295	3201	15000	0.500	https://www.openml.org/d/41161
25	fried	40768	10	20341	20427	5.000	https://www.openml.org/d/901
26	nomao	32062	118	22251	9811	5.000	The dataset has been enriched during the Nomao Challenge: organized along with the ALRA workshop
27	electricity	45312	8	19237	26075	5.000	This data was collected from the Australian New South Wales Electricity Market. In this market, prices are not fixed and are affected by demand and supply of the market. https://www.openml.org/d/151
28	Run_or_walk_information	88588	6	44365	44223	1.000	This dataset is gather to detect whether a person is running or walking based on deep neural networks and sensor data collected from iOS devices
29	vehicleNorm	98528	101	49264	49264	1.000	Vehicle classification in distributed sensor networks. https://www.openml.org/d/1242
30	numerai28.6	96320	21	48658	47662	1.000	Encrypted Stock Market Training Data from Numer.ai

Tables 6.3-6.4 provide the numerical results for SVM training with ALFPGM, NRAL, and an interior point method (IPM) native to MATLAB; *QuadProg*. Table 6.3 shows the results without the use of the decomposition method, that is, all data points are used simultaneously. In this case, the full $m \times m$ matrix Q is computed once and stored in memory as long as it can fit in memory. Table 6.4 shows the results of the SVM training using the selection method WSS_4 to test the performance of the 3 methods. For all of the dataset, the tables 6.4 and 6.5 show the number of training data points (m) and the number of features (n), the number of iterations ($nIter$) to get the SVM solution, training time, classification error in classifying the training dataset (Err) and a final objective function value, (Obj_{val}). Furthermore, the columns $nDcmp$ in Table 6.4 are the number of decomposition iterations,

that is, the number of selections of a working set and the subsequent SVM subproblem solution fining sequence. All simulation scenarios were averaged over 10 runs.

Fig. 6.1 shows the ratios of training times between ALFPGM and IPM without decomposition. Values smaller than 1.0, shown in red, indicate that the ALFPGM was faster, i.e. it took more time of the IPM to train an SVM for a particular data set, while values less than 1.0, shown in blue, indicate that the IPM was faster. Similarly, Fig. 6.2 shows the training times ratios between IPM and NRAL without decomposition. Fig. 6.3 shows the training times ratios between ALFPGM and NRAL. Fig. 6.4 shows the comparison of the training time histogram for IPM, ALFPGM, and NRAL.



Figure 6.1: Comparison between ALFPGM and IPM without decomposition. (MATLAB)

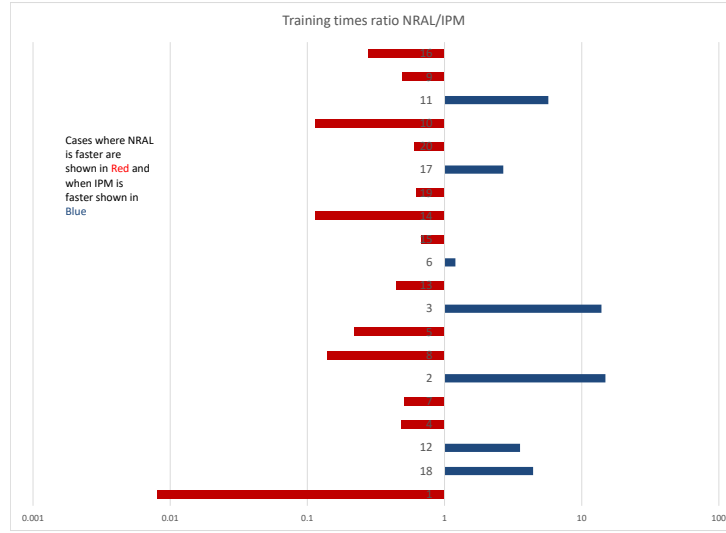


Figure 6.2: Comparison between IPM and NRAL without decomposition(MATLAB).

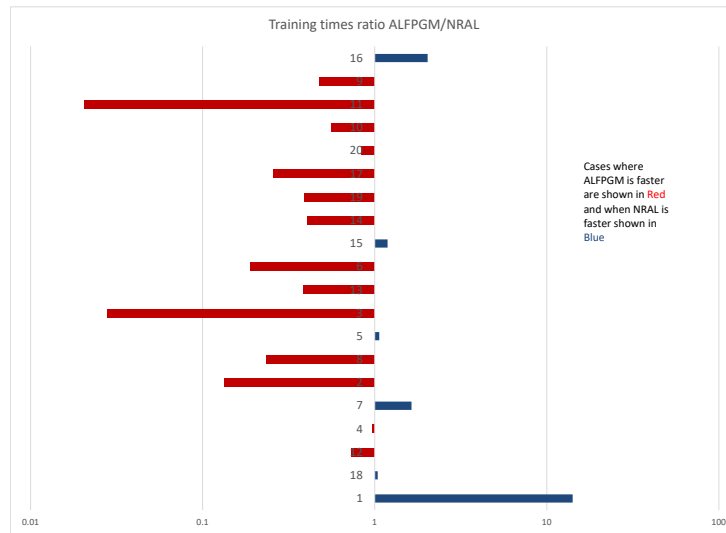


Figure 6.3: Comparison between ALFPGM and NRAL without decomposition (MATLAB).

Fig. 6.1 shows that ALFPGM outperforms QuadProg IPM for most training problems

without decomposition, especially when the number of training data is large. The cases where IPM was better, the time difference was in fractions of a second. Similarly, Fig. 6.2 shows that NRAL outperforms QuadProg IPM for most training sets without a decomposition scheme. As the Kernel Matrix gets larger, the condition number of the kernel matrix is important in determining the number of steps needed for IPM and NRAL iterations. It was observed that whenever the condition number of the matrix is large, i.e. the matrix is approaching singularity, it takes longer to find a solution using IPM and NRAL. Fig. 6.3 shows the comparison of the time between training times using NRAL and ALFPGM without a decomposition scheme. ALFPGM in most cases outperformed NRAL.

Table 6.5 shows the relative efficiency of the WSS_4 decomposition scheme in the absence of decomposition for the ALFPGM, NRAL and IPM. In all three methods, using decomposition, all the schemes showed reduced training times. Fig. 6.6 and 6.12 show the normalized training time, i.e., the training time divided by the number of training samples. The trend for 6.6 suggests that as the training samples grow larger, the normalized training time will increase. Fig. 6.12 on the other hand, has a flatter trend as the number of training data increases. This is a desired characteristic, as it means that we can train larger set SVM problems with an expected runtime that varies linearly with the number of training data. The results (Fig. 6.4-6.9) suggest that for training data sets within hundreds of points using no decomposition method is justified, while for larger problems using decomposition methods is preferable. This is of immense benefit since we can decompose large-set problems into smaller-set problems and solve each decomposition efficiently. This is the approach that will be used for the optimized high performance C++ implementation. Our focus will be on selecting the working set and efficiently solving the SVM subproblem.

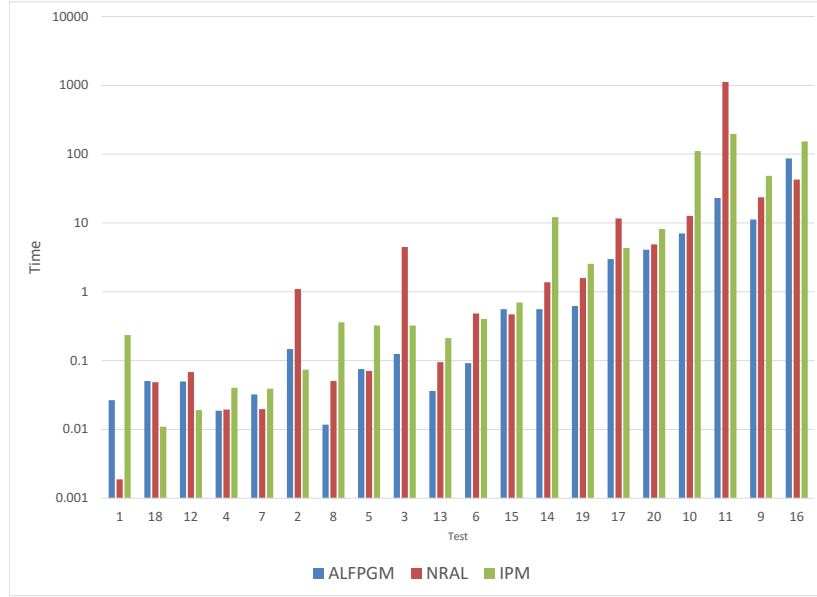


Figure 6.4: Training time comparison between ALFPGM, NRAL, and IPM without decomposition (MATLAB).

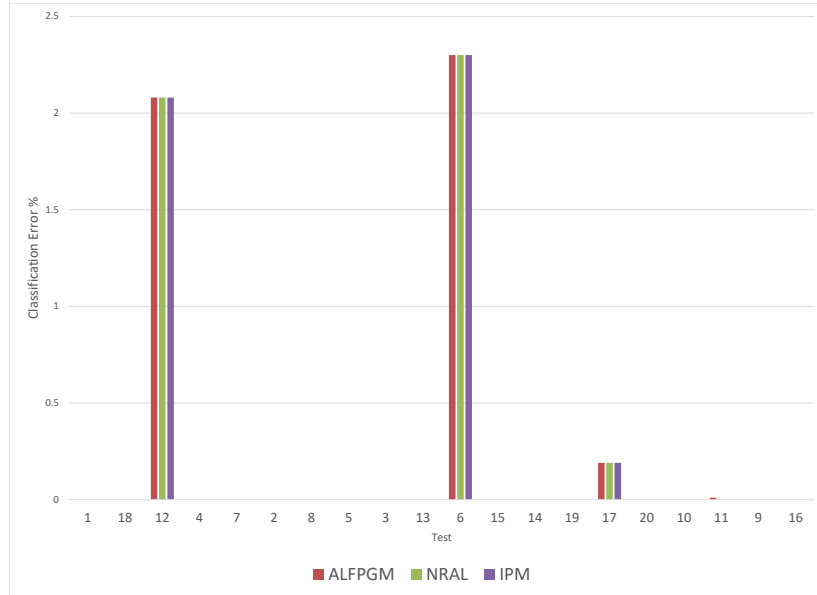


Figure 6.5: Comparison of non-decomposition classification errors (MATLAB).

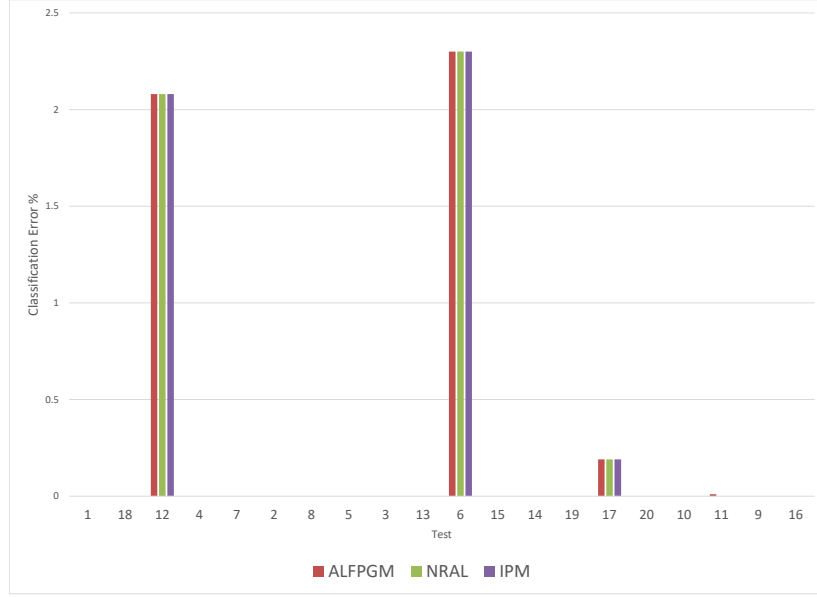


Figure 6.6: Comparison of normalized training time ALFPGM, NRAL, and IPM without decomposition (MATLAB).

Using decomposition gave smaller overall training times compared to training without decomposition. NRAL performed best in terms of overall training time compared to IPM or ALFPGM. When all 3 algorithms took similar number of decomposition iterations, NRAL performed best, then ALFPM overall. In most of the cases, NRAL outperformed ALFPGM. This is expected, as ALFPGM is a first-order method. The appeal of using the ALFPGM algorithm is its simplicity. In the next section, we will see how optimizing the implementation of the algorithm can reduce runtime. Part of our investigation includes finding ways to speed up the algorithm.

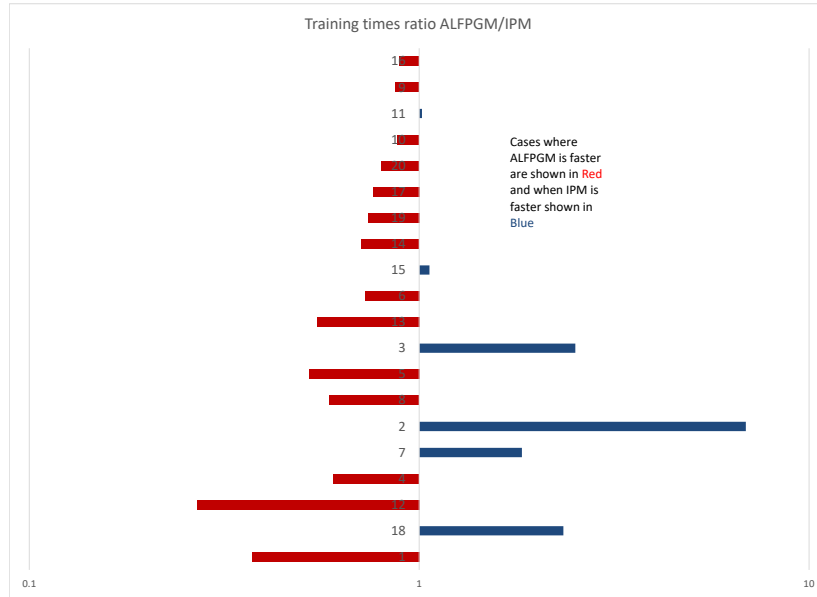


Figure 6.7: Comparison between ALFPGM and IPM with decomposition WSS_4 (MATLAB).

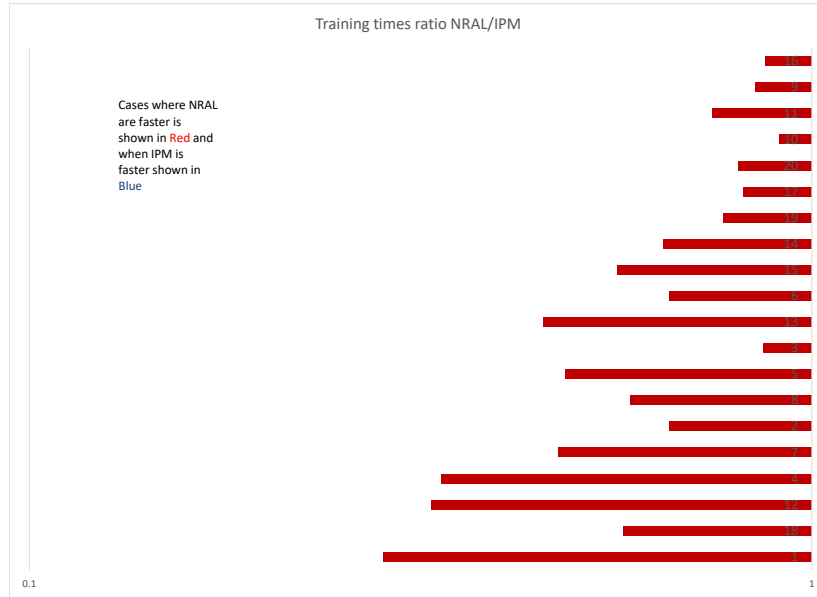


Figure 6.8: Comparison between IPM and NRAL with decomposition WSS_4 (MATLAB).

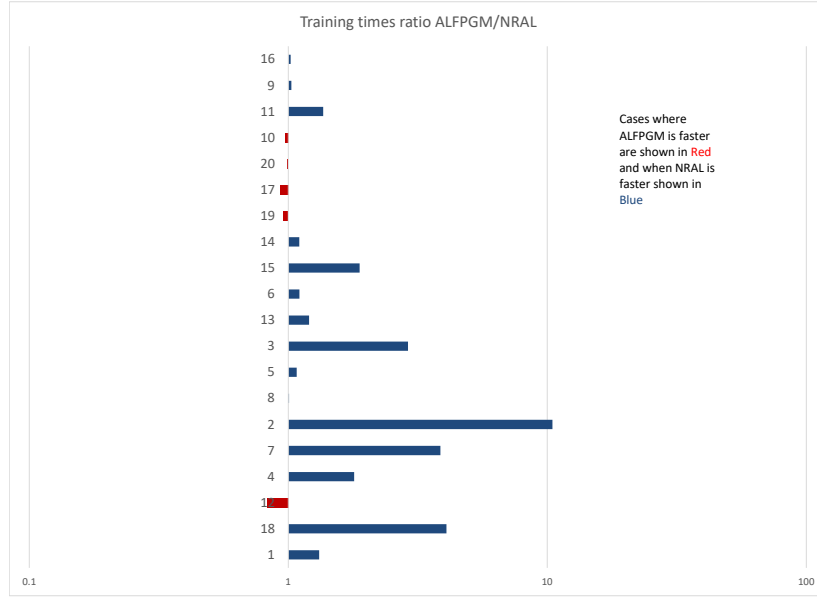


Figure 6.9: Comparison between ALFPGM and NRAL with decomposition WSS_4 (MATLAB).

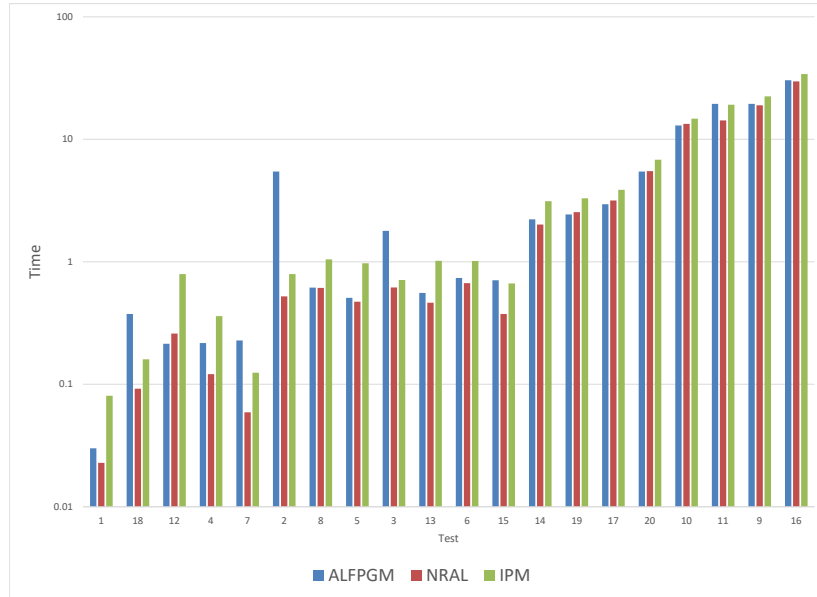


Figure 6.10: Comparison between training times with decomposition WSS_4 (MATLAB).

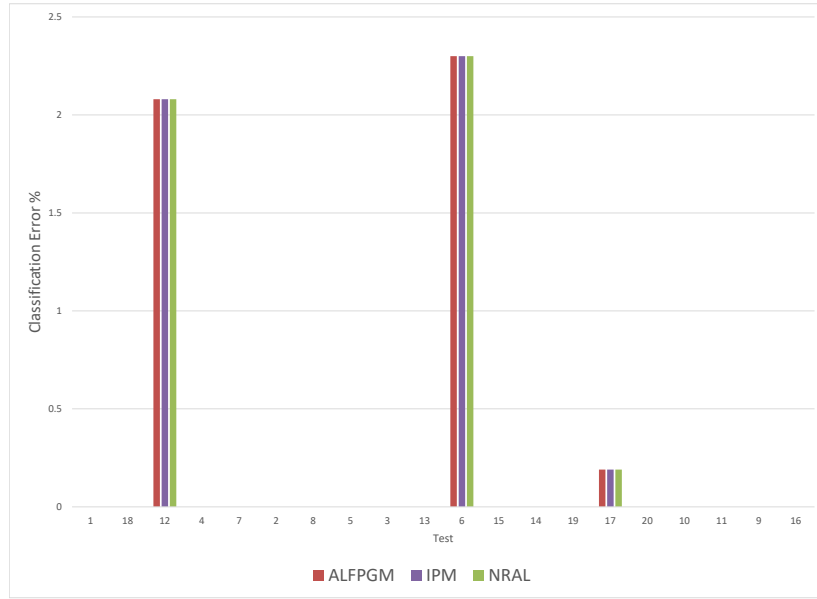


Figure 6.11: Comparison between classification errors (WSS_4).

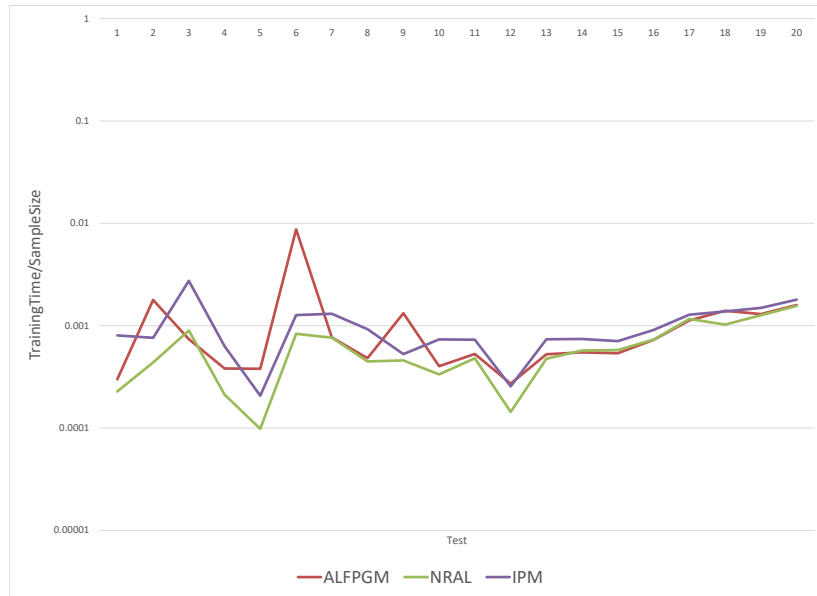


Figure 6.12: Normalized Training Time with Decomposition

C++ Implementations Results

With the promising results of the MATLAB simulations, the next step is C++ Implementations. The goal is to further reduce the training time using high performance C++ libraries and to run SVM on large set problems. The implementation will be optimized for larger-scale problems versus the rudimentary implementation done in MATLAB.

Comparison between different WSS schemes for ALFPGM

As mentioned above, the time to find the working set is often longer than the time to find the subsequent solution for the working set (see Fig. 6.13 - 6.15). In this section, we compare the timings for the different proposed WSS schemes; WSS_1 , WSS_2 , WSS_3 , WSS_5 , WSS_6 , WSS_7 and WSS_8 . $time_{WSS}$ is the time it takes for form the decomposition subproblems. $nDcmp$ is the number of decompositions required to meet the exit criteria. $time_{ALFPGM}$ is the time the algorithm spent solving the SVM subproblem.

Test 21 was used to benchmark the performance of all the working set selection schemes, as it is sufficiently sized. In all cases, the maximum number of decompositions was set to 10000. The following parameters were used $WSS_{minAlphaCheck} = \min(.1m, 100)$, $minAlphaOpt = 15$, $maxAlphaOpt = 20$. The results in Table 6.5 show that all the schemes gave the same training error. However, WSS_2 , WSS_3 , WSS_7 , and WSS_8 have a different final objective value. WSS_0 performed slightly better compared to WSS_4 as it does not need to perform all the kernel computations necessary for the second-order WSS schemes. For some problems like the case we have here, it is okay to exit the decomposition iterations early. WSS_2 , WSS_3 , WSS_7 and WSS_8 gave a similar classification error as the other schemes (WSS_1 , WSS_4 and WSS_5) and the computation was stopped earlier. That is, by limiting the number of optimizations α , we can achieve a faster training time while maintaining the same classification error percentage. Fig. 6.15 shows the comparison between $time_{WSS}$, the time to find the working set, and $time_{ALFPGM}$, the time to find the subsequent SVM subproblem solution. Fig. 6.16 shows the comparison between the number of outer iterations

(Augmented Langragian iterations) and the inner FPGM iterations. The number of inner iterations is relative to the number of outer iteration calls, which depends on the number of decomposition rounds. These data suggest that by reducing the number of decompositions needed, as we have done, we can reduce the overall training time.

Table 6.5: WSS timing versus ALFPGM timing for test 21 (100 pairs)

WSS_{type}	nD	$time_{WSS}$	$time_{ALFPGM}$	$nInnIter$	$nOutIter$	$nWss$	Err	Obj_{val}	Time s
WSS_0	1668	98.0722	2.445647	170819	5509	200	4.59	-350686	103.9413
WSS_1	1682	92.17027	2.368635	171071	5520	200	4.59	-350686	97.81993
WSS_2	1695	95.40773	2.358085	169706	5713	200	4.59	-347960	101.0755
WSS_3	1696	95.61518	2.445585	169725	5717	199.9268	4.59	-347960	101.4111
WSS_4	1734	89.87098	2.251358	172270	5762	200	4.59	-350686	95.49283
WSS_5	1652	90.21417	2.363027	170275	5320	200	4.59	-350686	95.88432
WSS_6	1664	87.27448	2.316799	170456	5454	200	4.59	-350686	92.79675
WSS_7	1709	91.0776	2.308469	170065	5815	200	4.59	-347894	96.75294
WSS_8	1710	87.75037	2.492563	170084	5819	199.9263	4.59	-347894	93.58626

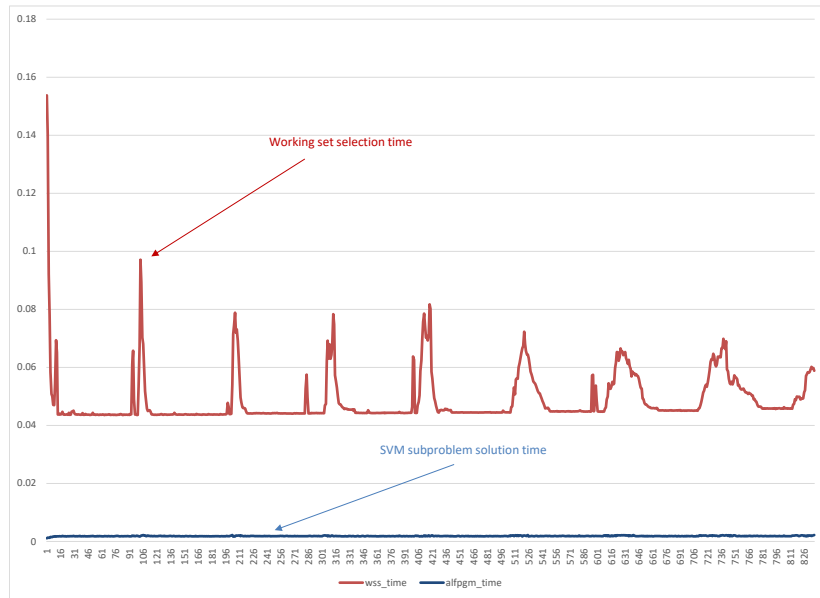


Figure 6.13: Comparison between WSS time and ALFPGM time for webdata.wXa WSS_0

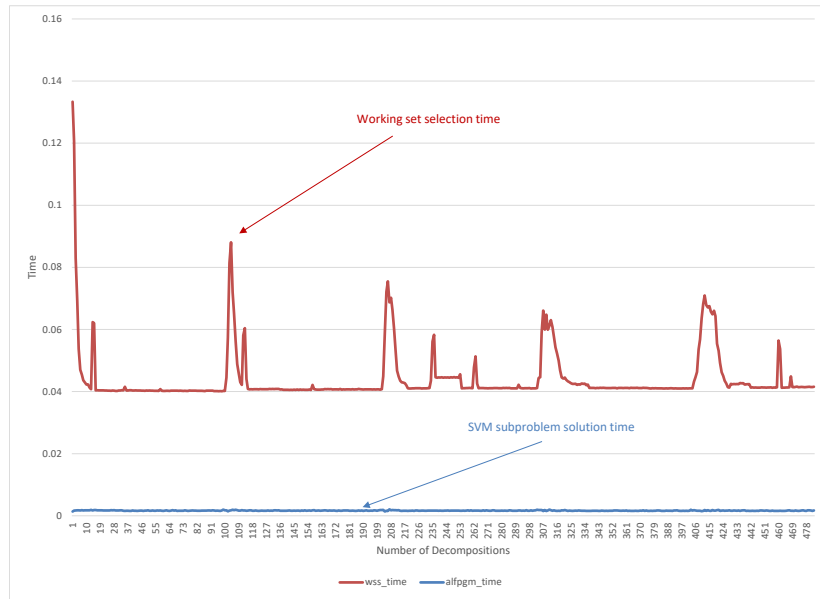


Figure 6.14: Comparison between WSS time and ALFPGM time for webdata_wXa WSS_4

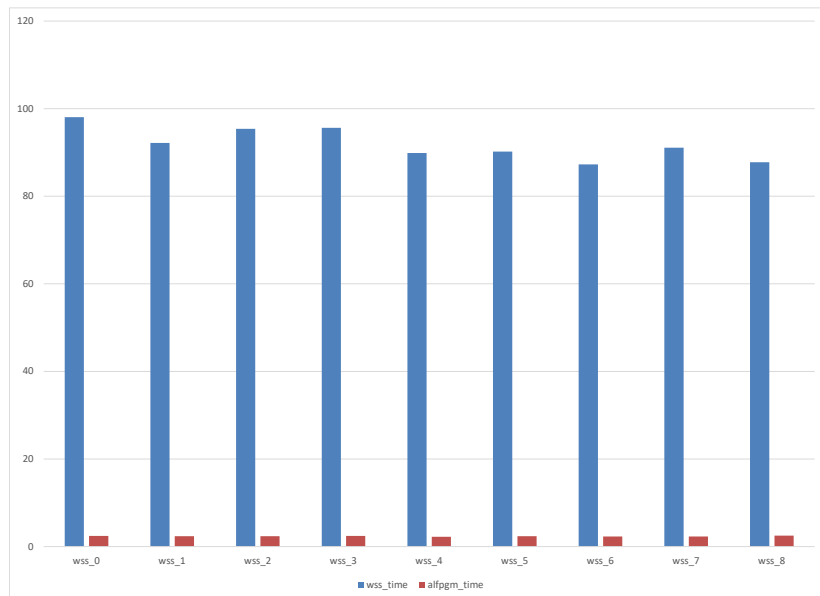


Figure 6.15: Comparison between WSS types - timings

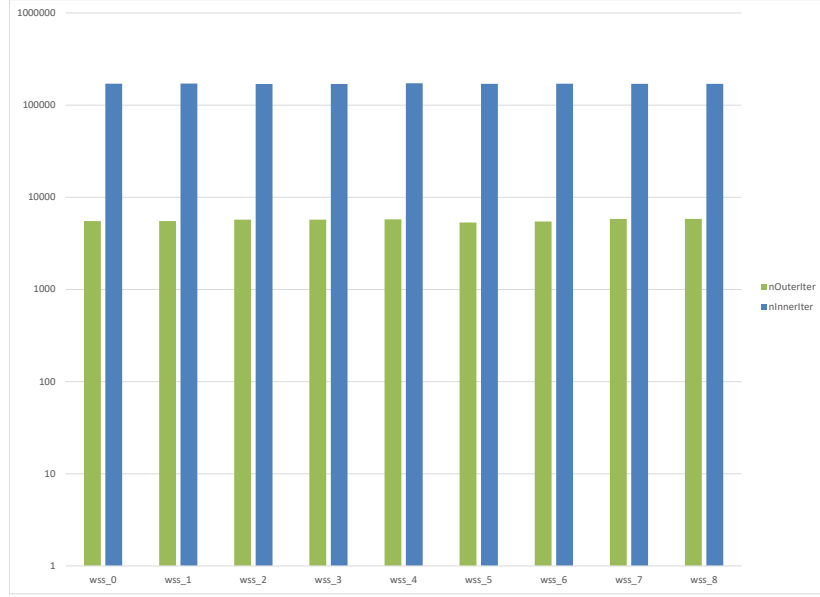


Figure 6.16: Comparison between WSS types - Total FPGM Iterations

Table 6.6: WSS timing versus ALFPGM timing for Test 28 (100 pairs)

WSS_{type}	nD	$time_{WSS}$	$time_{ALFPGM}$	$nInnIter$	$nOutIter$	$nWss$	Err	Obj_{val}	Time s
WSS_0	6370	918.1699	37.35849	3442773	31736	200	0.03	-26753.4	981.8407
WSS_1	2274	549.8224	9.955537	928624	11284	197.5117	0.07	-23741.6	569.4888
WSS_2	2250	452.4246	10.194	922860	11188	198.5273	0.07	-23741.5	472.114
WSS_3	2258	512.6824	12.18969	894725	11190	197.9078	0.06	-23746.8	534.1555
WSS_4	4962	648.3794	26.11965	2511810	24530	200	0.03	-26753.4	694.6812
WSS_5	4970	639.154	27.17421	2543404	25452	195.8962	0.03	-26753.4	686.6167
WSS_6	2274	319.5706	9.654415	897438	11261	197.4483	0.06	-24475.2	338.6523
WSS_7	2253	326.5483	9.913783	950054	11340	198.1137	0.06	-24473.1	345.7599
WSS_8	2267	312.8918	12.22372	901042	11387	196.0733	0.06	-24383.8	334.4376

Another test was performed using Test 28. Similarly, $WSS_{minAlphaCheck} = \min(.1m, 100)$, $minAlphaOpt = 150$, $maxAlphaOpt = 200$. Unlike test 21, tests WSS_0 , WSS_4 and WSS_5 in Test 28 (Table 6.6) take a long time and iterations to arrive at the right solution before stopping. WSS_1 , WSS_2 , WSS_3 , WSS_7 and WSS_8 have the fastest times with a similar

classification error (0.05 vs. 0.03). $minAlphaOpt$ and $maxAlphaOpt$ need to be selected based on the total number of sample data m and the number of pairs p . This is particularly important for large dataset problems. It is important to ensure that α are well tested with other α values before they are no longer used in the WSS process. A good number of start with is $minAlphaOpt = \max(5, \frac{m}{10p})$, $maxAlphaOpt = \max(15, \frac{m}{10p})$. Table 6.6 suggests that for very large data set problems where there are many decomposition steps and $minAlphaOpt$ are large, we can get faster training time using WSS_2 , WSS_3 , WSS_7 or WSS_8 .

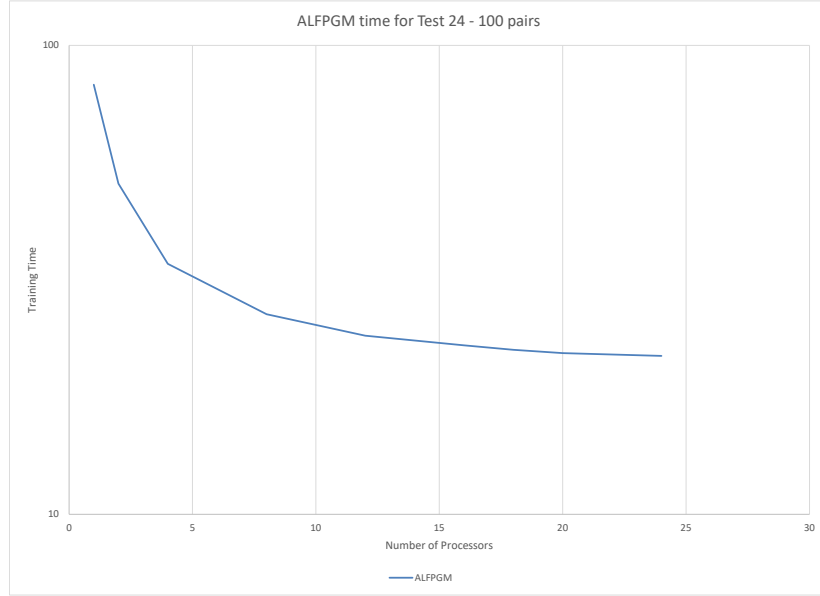


Figure 6.17: Performance improvements with using Multiprocessing Test 24 (18201 data points 4295 features)

ALFPGM results improvements with multiprocessing

We investigated the scalability performance of ALFPGM. We compared the training time with the number of processors used for the training. Fig. 6.17 shows the training time

versus the number of processor threads used in the training. The biggest gains were up to using 12 processors, and then there were diminishing returns. This is an expected trend in parallel programming. There is a major benefit in going from one processor to two; there is a not as great but still substantial benefit in going from two processors to four. However after a number of processors, the real-time reduction in adding more processors is very small compared to the substantial increase in resource consumption.

ALFPGM - Comparison of using different number of pairs p

In this section we tried to find the optimum p to use for ALFPGM. We used Tests 16, 25 and 30 and varied the number of pairs. In all cases, we used WSS_7 . The results are presented in Tables 6.7 - 6.9 and Fig. 6.18 - 6.20. The results suggest that for ALFPGM the optimum number of pairs is $100 \leq p \leq 20$.

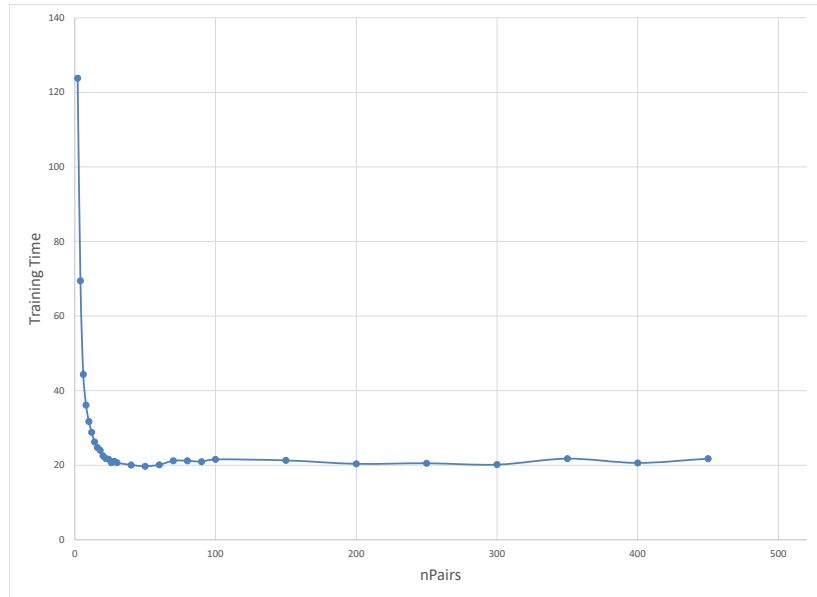


Figure 6.18: ALFPGM - Classification time for different number of pairs p - Test 16.

There are several factors that come into play when varying the number of pairs p .

1. The size of p determines the amount of time it takes to find the size of the working set. The larger the size of p , the longer it takes to find the working set.
2. The smaller the size of p , the more the number of decompositions needed to solve the SVM problem.
3. The larger the size of p , the longer it takes to solve the SVM subproblems.
4. The larger the size of p , the greater the probability that the Hessian matrix of the SVM subproblem may be degenerate.

Table 6.7: ALFPGM results for Test 16 with varying nPairs

<i>nPairs</i>	ALFPGM Time					
	<i>Time</i> (s)	<i>Err</i>	<i>nSv</i>	<i>Obj_{val}</i>	<i>Iterations</i>	<i>nD</i>
2	123.804721	0	18905	-8577.31	418040	30973
4	69.483926	0	18905	-8577.31	140079	15327
6	44.3587575	0	18905	-8577.31	74424	10127
8	36.1464555	0	18905	-8577.31	48375	7600
10	31.739852	0	18905	-8577.31	28573	6018
12	28.865686	0	18905	-8577.31	25424	5036
14	26.272481	0	18905	-8577.31	23297	4273
16	24.756381	0	18905	-8577.31	26796	3722
18	24.0292085	0	18905	-8577.31	15480	3374
20	22.5715785	0	18905	-8577.31	13764	3001
22	21.8276955	0	18905	-8577.31	12206	2671
24	21.6393575	0	18905	-8577.31	11374	2480
26	20.6966755	0	18905	-8577.31	10510	2296
28	21.091971	0	18905	-8577.31	10067	2193
30	20.6824865	0	18905	-8577.31	9318	2030
40	20.09519	0	18905	-8577.31	7030	1533
50	19.734837	0	18905	-8577.31	5657	1238
60	20.1373165	0	18905	-8577.31	5611	1031
70	21.220922	0	18905	-8577.31	3529	876
80	21.2192915	0	18905	-8577.31	2798	750
90	20.9901475	0	18905	-8577.31	2484	660
100	21.591046	0	18905	-8577.31	2370	614
150	21.3120485	0	18905	-8577.31	1469	388
200	20.3956925	0	18905	-8577.31	1162	279
250	20.5481675	0	18905	-8577.31	1000	219
300	20.1807495	0	18905	-8577.31	929	183
350	21.7772445	0	18905	-8577.31	1251	166
400	20.6398145	0	18905	-8577.31	1314	132
450	21.7796	0	18905	-8577.31	1280	118
500	21.074534	0	18905	-8577.31	1774	105

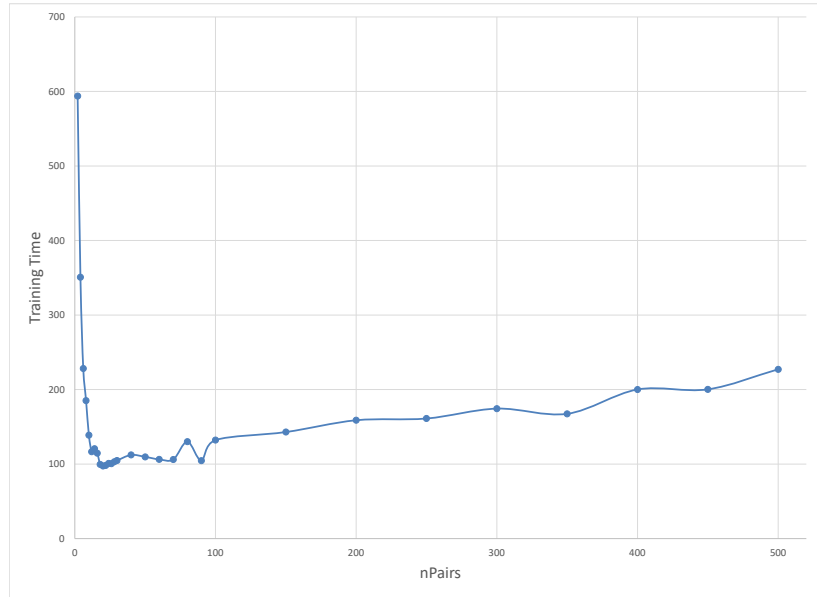


Figure 6.19: ALFPGM - Classification time for different numbers of pairs p Test 25.

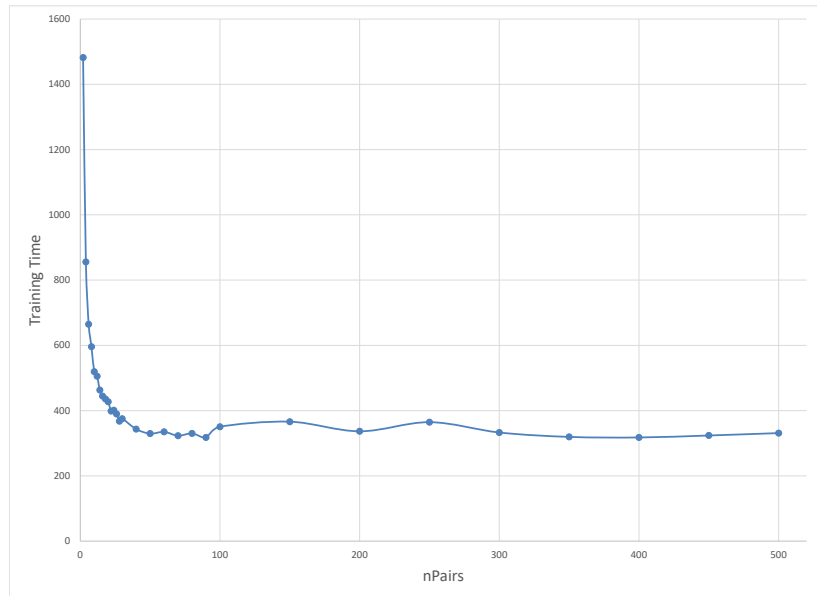


Figure 6.20: ALFPGM - Classification time for different numbers of pairs p - Test 30.

Table 6.8: ALFPGM results for Test 25 with varying nPairs

<i>nPairs</i>	ALFPGM Time					
	<i>Time</i> (s)	<i>Err</i>	<i>nSv</i>	<i>Obj_{val}</i>	<i>Iterations</i>	<i>nD</i>
2	593.8708415	0.05	10867	-15727.2	1206607	99430
4	350.6830495	0.09	10872	-16725.6	475283	54164
6	228.1724265	0.03	10945	-16610.2	228739	31475
8	185.2587545	0.02	10969	-16817.8	148488	22825
10	138.738265	0	11117	-17598.1	96789	15919
12	116.5954155	0	11132	-17592.2	71915	12367
14	120.585538	0	11121	-17598.2	66548	11795
16	114.68055	0	11124	-17598.1	57884	10494
18	99.606217	0	11132	-17581.7	45585	8602
20	97.4593705	0	11138	-17580.3	40768	7855
22	98.276515	0	11136	-17579.1	36672	7232
24	101.0070185	0	11144	-17572.6	33741	6756
26	100.576808	0	11161	-17567.5	31190	6321
28	103.105102	0	11164	-17566.5	29144	5909
30	104.710064	0	11158	-17555.9	27015	5494
40	112.2581095	0	11174	-17536.7	21195	4334
50	109.6898605	0	11225	-17512.2	17002	3502
60	106.2746085	0	11227	-17478.1	14355	2982
70	106.137575	0	11245	-17456.8	12239	2592
80	130.1329695	0	11141	-17573.5	13302	2875
90	104.638106	0	11273	-17394.3	9013	2050
100	132.1765945	0	11160	-17560.4	10448	2394
150	142.9638765	0	11200	-17537.3	7043	1722
200	158.803399	0	11225	-17490.2	5824	1445
250	161.12544	0	11230	-17444.3	4805	1177
300	174.417873	0	11254	-17406.2	4343	1028
350	167.45417	0	11271	-17375.4	3984	873
400	200.115672	0	11270	-17364.7	4467	838
450	200.303497	0	11303	-17342.4	5234	765
500	227.1943725	0	11329	-17333.9	6730	756

Table 6.9: ALFPGM results for Test 30 with varying nPairs

<i>nPairs</i>	ALFPGM Time					
	<i>Time</i> (s)	<i>Err</i>	<i>nSv</i>	<i>Obj_{val}</i>	<i>Iterations</i>	<i>nD</i>
2	1481.921126	0	96300	-49857.5	328275	70958
4	855.79962	0	96304	-49916.9	130546	34978
6	664.7104145	0	96304	-49916.9	78143	23214
8	595.7563115	0	96304	-49916.9	54384	17532
10	519.3031835	0	96304	-49916.9	32714	14060
12	505.133634	0	96304	-49916.9	26678	11738
14	463.1722295	0	96304	-49916.9	23418	10160
16	444.3332525	0	96304	-49916.9	21237	8870
18	435.456936	0	96302	-49912.7	20293	8083
20	427.0975085	0	96304	-49916.9	16641	7335
22	398.6321075	0	96304	-49916.9	14542	6576
24	401.0065995	0	96304	-49916.9	13165	6028
26	389.885153	0	96304	-49916.9	12138	5638
28	367.571654	0	96304	-49916.9	11087	5167
30	375.2423985	0	96304	-49916.9	10624	4961
40	343.3429125	0	96304	-49916.9	7837	3684
50	329.586473	0	96304	-49916.9	6323	2928
60	334.875853	0	96304	-49916.9	5439	2545
70	323.034733	0	96304	-49916.9	4848	2225
80	330.184603	0	96304	-49916.9	4272	1973
90	317.8700565	0	96304	-49916.9	3752	1706
100	350.464458	0	96304	-49916.9	3593	1653
150	366.0834735	0	96304	-49916.9	2659	1226
200	336.5929005	0	96304	-49916.9	1902	871
250	364.5629655	0	96304	-49916.9	1644	743
300	332.916956	0	96304	-49916.9	1398	555
350	319.4398085	0	96304	-49916.9	1562	466
400	317.690327	0	96304	-49916.9	1922	411
450	323.781645	0	96304	-49916.9	2584	365
500	331.097487	0	96304	-49916.9	3397	331

Next, we present results for different number of working set pairs; 10, 50, 100, 200 and 250 for small to mid-sized SVM problems (Tests 1-24). It can be stated that in terms of few iterations and reduced classification errors, WSS_2 , WSS_3 , WSS_7 and WSS_8 are the desired working set selection schemes to use. Figs. 6.21 - 6.22 further show that WSS_2 , WSS_3 , WSS_7 and WSS_8 performed the best in terms of training classification errors and training times. WSS_0 and WSS_5 clearly performed the worst. It is also observed that for ALFPGM, the number of decompositions reduces with the number of pairs used. However, the more pairs used, the more the time used in computing the working set. It is important to

strike a good balance between the choice of the number of decompositions and the number of points. The relationship between them is observed to be inversely related. We have observed that for ALFPGM from Figs. 6.18 - 6.21, $100 \leq p \leq 250$ are good choices for p for the test problems when using WSS_2 , WSS_3 , WSS_7 and WSS_8 .

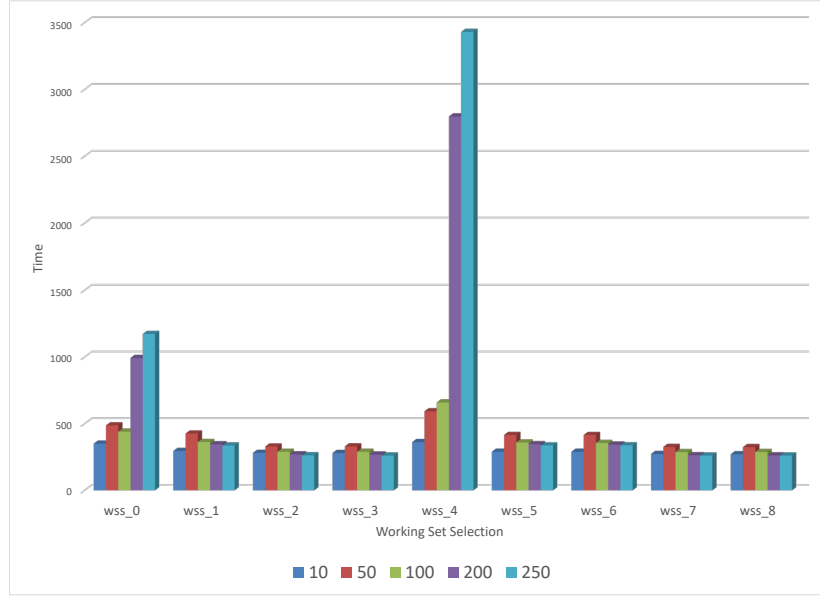


Figure 6.21: Training time comparison between WSS types and nPairs p

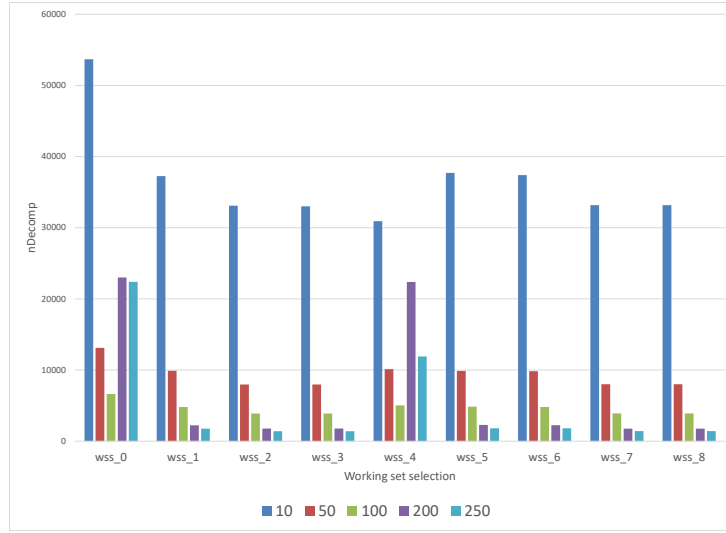


Figure 6.22: Comparison of the total number of decompositions between WSS types and nPairs p

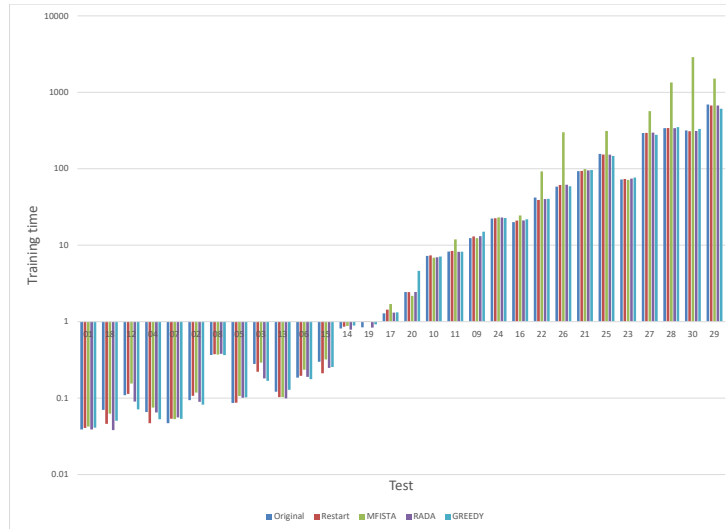


Figure 6.23: Comparison between Training times Fista types and the number of pairs - WSS_7 Time

Comparison between different Inner FPGM Procedures

The next section to optimize after selecting the working set, is to efficiently solve the SVM subproblem in the fastest amount of overall training time and classification errors. In this section, we present results of using different FISTA algorithm discussed in Algorithms 1 - 5.

Results from Table 6.10 and Fig. 6.23 show the performance difference between the different inner FISTA variants. There's a significant performance loss by using MFISTA chiefly due to the time cost of computing $F(z_k)$. Although there isn't much difference between other FISTA variants, GREEDY gives the least training time overall and then RESTART FISTA. It can be concluded that GREEDY FISTA is preferred to other FISTA methods as it provides the fastest training time and classification time.

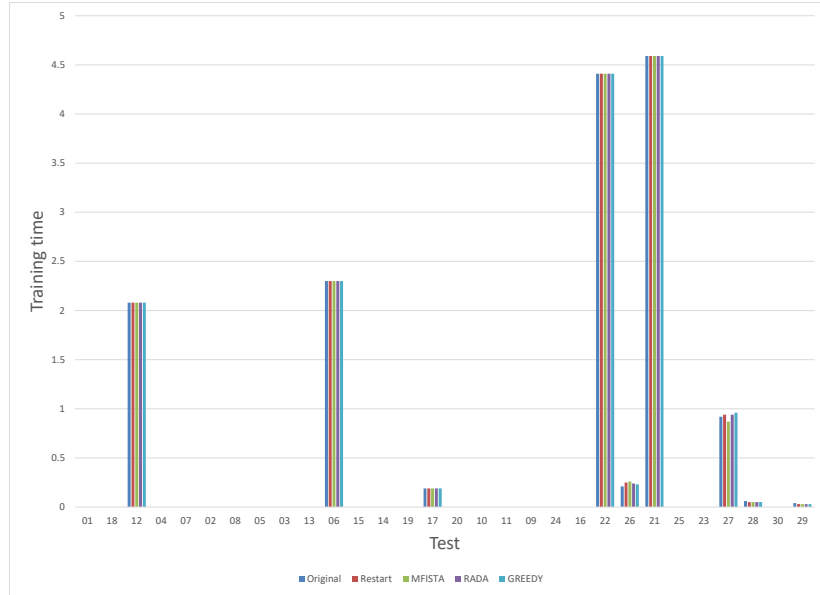


Figure 6.24: Comparison between Fista types - Classification errors - Time (100 Pairs)

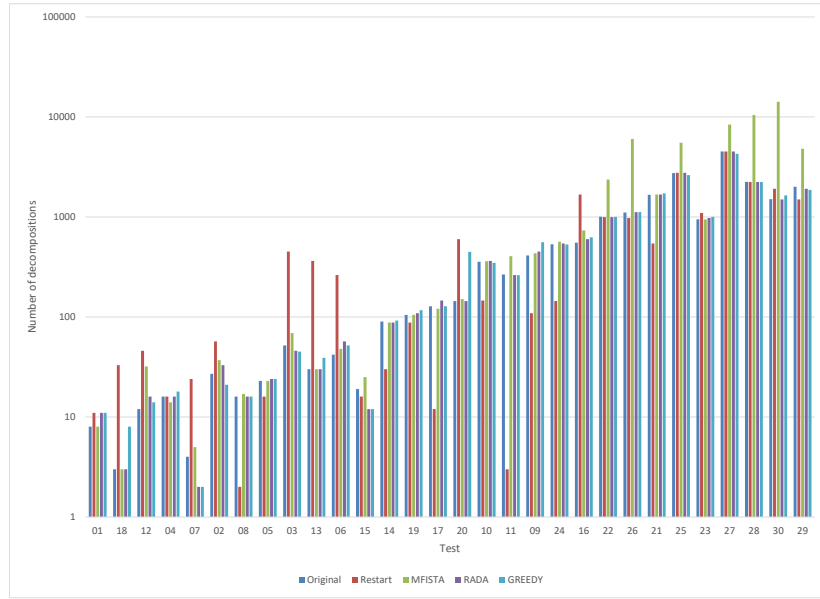


Figure 6.25: Comparison between Fista types - Number of decompositions - Time (100 Pairs)

Table 6.10: Comparing different FISTA methods for WSS_8 (100 Pairs)

test	FISTA			RESTART			MFISTA			RADA			GREEDY		
	Time (s)	Obj _{val}	Err	Time (s)	Obj _{val}	Err	Time (s)	Obj _{val}	Err	Time (s)	Obj _{val}	Err	time	Obj _{val}	Err
01	0.038913	-49.2803	0	0.040754	-49.2808	0	0.042495	-49.2803	0	0.038957	-49.2808	0	0.041081	-49.2799	0
02	0.094174	-359.791	0	0.107219	-359.791	0	0.117858	-359.79	0	0.08952	-359.791	0	0.082175	-359.791	0
03	0.280476	-68.5122	0	0.221626	-68.5129	0	0.291171	-68.5083	0	0.181359	-68.5129	0	0.168243	-68.5124	0
04	0.065592	-262.212	0	0.047023	-262.213	0	0.075165	-262.211	0	0.064874	-262.213	0	0.053013	-262.213	0
05	0.086324	-199.878	0	0.086981	-199.877	0	0.106515	-199.869	0	0.101312	-199.877	0	0.102261	-199.877	0
06	0.185189	-7043.52	2.3	0.195265	-7043.52	2.3	0.23533	-7043.52	2.3	0.189998	-7043.52	2.3	0.176839	-7043.52	2.3
07	0.046937	-300	0	0.053766	-300	0	0.053127	-300	0	0.055799	-300	0	0.053598	-300	0
08	0.366688	-140.793	0	0.375746	-140.793	0	0.371146	-140.793	0	0.380068	-140.793	0	0.365688	-140.787	0
09	12.38959	-7411.46	0	12.99889	-7411.46	0	12.39363	-7411.46	0	13.10302	-7411.46	0	15.03442	-7411.46	0
10	7.260673	-3679.4	0	7.368565	-3679.4	0	6.858477	-3679.4	0	6.946436	-3679.4	0	7.097385	-3679.39	0
11	8.274541	-915.154	0	8.434529	-916.522	0	11.92472	-914.512	0	8.202951	-916.522	0	8.244953	-916.586	0
12	0.1094	-1340.6	2.08	0.113673	-1340.6	2.08	0.155337	-1340.6	2.08	0.090144	-1340.6	2.08	0.071357	-1340.6	2.08
13	0.121414	-691.624	0	0.103295	-691.624	0	0.10302	-691.624	0	0.099121	-691.624	0	0.128592	-691.623	0
14	0.814571	-2110.77	0	0.860763	-2110.77	0	0.878468	-2110.77	0	0.78531	-2110.77	0	0.891187	-2110.77	0
15	0.298805	-1300	0	0.211301	-1300	0	0.320753	-1300	0	0.248542	-1300	0	0.255422	-1300	0
16	20.05606	-8577.31	0	21.02337	-8577.31	0	24.461	-8577.31	0	21.04783	-8577.31	0	21.79656	-8577.31	0
17	1.283186	-2912.07	0.19	1.43522	-2912.07	0.19	1.701872	-2912.07	0.19	1.312119	-2912.07	0.19	1.323105	-2912.07	0.19
18	0.070047	-58.4939	0	0.04613	-58.4939	0	0.062771	-58.4939	0	0.038173	-58.4939	0	0.050559	-58.4947	0
19	0.84457	-1505.53	0	1.015225	-1505.53	0	1.01632	-1505.53	0	0.835778	-1505.53	0	0.922081	-1505.53	0
20	2.444329	-3746.73	0	2.43843	-3746.73	0	2.179502	-3746.73	0	2.448114	-3746.73	0	4.622349	-3746.73	0
21	92.96106	-350686	4.59	93.5767	-350686	4.59	98.76583	-350686	4.59	95.31999	-350686	4.59	96.48033	-350686	4.59
22	42.09455	-244605	4.41	39.0132	-244605	4.41	92.26947	-244605	4.41	40.12986	-244605	4.41	40.43334	-244605	4.41
23	72.3855	-9340.53	0	73.59288	-9340.53	0	71.61239	-9340.53	0	74.24932	-9340.53	0	76.75029	-9340.53	0
24	22.35385	-5175.66	0	22.49883	-5175.66	0	23.09469	-5175.66	0	23.00128	-5175.66	0	22.69035	-5175.66	0
25	156.701	-17589	0	153.4727	-17589.7	0	311.7061	-17591.2	0	152.8625	-17589.7	0	147.329	-17589.4	0
26	58.34985	-14985.7	0.21	61.38474	-15113.7	0.25	299.8895	-15296.9	0.26	61.81082	-15113.6	0.24	58.9206	-14998.5	0.23
27	293.0846	-118561	0.92	294.5178	-117007	0.94	567.5985	-122388	0.87	297.2349	-117007	0.94	277.5565	-117291	0.96
28	339.9139	-24450.8	0.06	340.5446	-24120.3	0.05	1347.086	-24309.5	0.05	339.6055	-24120.3	0.05	350.0972	-24158.3	0.05
29	694.025	-56318.2	0.04	671.1586	-56477.5	0.03	1507.994	-56625.3	0.03	672.7025	-56477.5	0.03	611.2381	-56667.1	0.03
30	316.7312	-49916.9	0	309.6402	-49916.9	0	2882.506	-49916.9	0	311.5226	-49916.9	0	332.3757	-49916.9	0
	2143.732			2116.578			7265.872			2124.699			2075.352		

NRAL-Newton results

We performed a similar experiment done using ALFPGM to train Test 21 and 28 using the NRAL algorithm. Tables 6.11 and 6.12 again demonstrate that WSS_2 , WSS_3 , WSS_7 and WSS_8 performed best in terms of training errors and training times. Results in Table 6.11 show that WSS schemes WSS_0 , WSS_4 and WSS_5 were unable to train the data in Test 21 in 20,000 decomposition steps. For the rest of the analysis, WSS_2 , WSS_3 , WSS_6 , WSS_7 and WSS_8 also performed well in experiments with Test 28.

Table 6.11: NRAL-Newton results for Test 21 (100 pairs)

WSS_{type}	nD	$time_{WSS}$	$time_{ALFPGM}$	$nIter$	$nWss$	Err	Obj_{val}	Time s
WSS_0	20001	868.3539	109.3493	179275	200	23.41	-60663.5	1013.191
WSS_1	1591	90.24955	4.812982	7763	200	4.59	-350686	98.35881
WSS_2	1591	87.93712	4.801774	7763	200	4.59	-350686	95.93165
WSS_3	1630	100.0099	5.0227	7875	199.9091	4.59	-350686	108.3271
WSS_4	20001	801.6258	102.4543	179266	200	23.41	-59880.9	939.5642
WSS_5	20001	812.2638	101.73	179253	200	23.07	-101701	949.0619
WSS_6	1746	88.81728	5.068855	7992	200	4.59	-350686	97.17176
WSS_7	1746	88.39174	5.123011	7992	200	4.59	-350686	96.82016
WSS_8	1710	90.72546	5.086147	7932	199.9684	4.59	-350686	99.15875

Table 6.12: NRAL-Newton results for Test 28 (100 pairs)

WSS_{type}	nD	$time_{WSS}$	$time_{ALFPGM}$	$nIter$	$nWss$	Err	Obj_{val}	Time s
WSS_0	20001	2291.143	295.0557	510747	200	0.03	-26221.2	2669.789
WSS_1	2280	670.9629	30.76834	55159	195.0031	0.03	-25678.3	711.1401
WSS_2	2228	452.9826	30.29251	54536	198.6053	0.03	-25678.3	492.5575
WSS_3	2590	712.2013	30.02383	53216	196.4828	0.03	-25722.7	752.7499
WSS_4	20001	2306.779	367.7518	707220	200	0.03	-26753.4	2754.798
WSS_5	7347	929.5055	127.3393	233074	199.9088	0.03	-26753.4	1086.555
WSS_6	2285	316.9574	28.38766	50332	199.2811	0.03	-25679.6	354.8905
WSS_7	2289	339.0514	27.7065	50456	199.3024	0.03	-25679.6	376.2853
WSS_8	2289	330.2781	29.74468	53514	199.0009	0.03	-25750.8	369.5336

NRAL-Newton results improvements with multiprocessing

We investigated the performance of NRAL with different number of processors. Fig. 6.26 shows the training time versus the number of processor threads used in the training. In all number of processor cases, we used WSS_7 . The biggest gains similar to using ALFPGM were up to using 12 processors, and then there were diminishing returns as expected while using parallel programming.

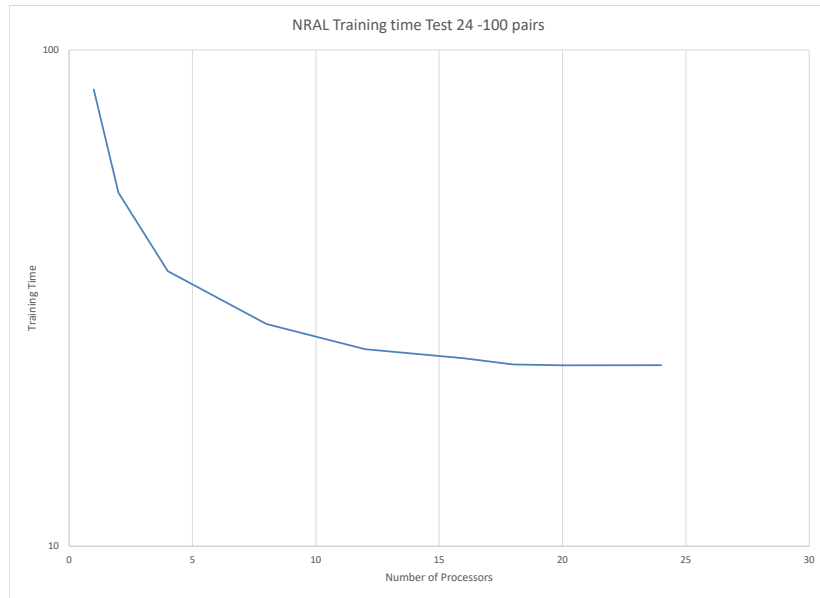


Figure 6.26: Performance improvements with using Multiprocessing Test 24 (18201 data points 4295 features)

NRAL - Comparison of using different number of pairs p

In this section we tried to find the optimum p to use for NRAL. We used Tests 16, 25 and 30 and varied the number of pairs. In all cases, we used WSS_7 . The results are presented in Tables 6.13 - 6.15 and Fig. 6.28 - 6.29. The results suggest that for NRAL the optimum number of pairs is between 10 and 50. In Fig. 6.27, the training time increased when

$p > 150$, this is because it took longer to solve the linear system needed by the Newton method as the subproblem matrices became larger. For Test 25, the **condition number** of the matrices was larger with larger p .

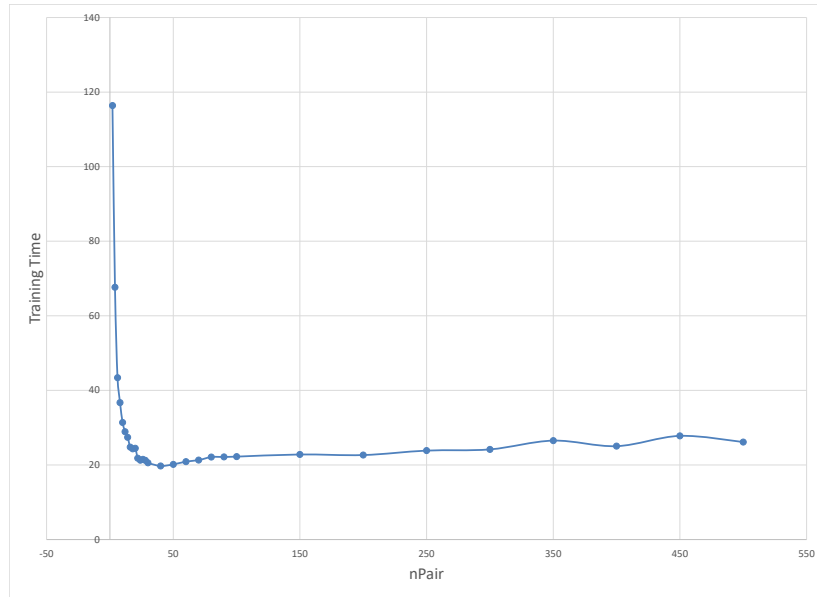


Figure 6.27: NRAL - Classification time for different pairs - Test 16.

Table 6.13: NRAL-Newton results for Test 16 with varying nPairs

<i>nPairs</i>	NRAL					
	<i>Time</i> (s)	<i>Err</i>	<i>nSv</i>	<i>Obj_{val}</i>	<i>Iterations</i>	<i>nD</i>
2	116.402254	0	18905	-8577.31	58646	30945
4	67.644663	0	18905	-8577.31	30681	15338
6	43.397424	0	18905	-8577.31	20241	10119
8	36.709577	0	18905	-8577.31	15086	7542
10	31.384651	0	18905	-8577.31	12020	6009
12	28.915261	0	18905	-8577.31	9849	4921
14	27.437907	0	18905	-8577.31	8570	4284
16	24.798495	0	18905	-8577.31	7483	3740
18	24.320789	0	18905	-8577.31	6721	3359
20	24.503561	0	18905	-8577.31	6000	2999
22	21.802208	0	18905	-8577.31	5464	2731
24	21.290395	0	18905	-8577.31	5062	2530
26	21.527157	0	18905	-8577.31	4624	2311
28	21.244577	0	18905	-8577.31	4364	2181
30	20.5716	0	18905	-8577.31	4122	2060
40	19.745366	0	18905	-8577.31	3060	1529
50	20.1519	0	18905	-8577.31	2488	1243
60	20.906937	0	18905	-8577.31	2052	1025
70	21.302046	0	18905	-8577.31	1777	887
80	22.127741	0	18905	-8577.31	1532	765
90	22.156554	0	18905	-8577.31	1374	686
100	22.237453	0	18905	-8577.31	1209	602
150	22.818557	0	18905	-8577.31	819	396
200	22.6635	0	18905	-8577.31	616	296
250	23.853368	0	18905	-8577.31	489	233
300	24.181126	0	18905	-8577.31	413	194
350	26.533043	0	18905	-8577.31	357	167
400	25.048355	0	18905	-8577.31	295	135
450	27.806044	0	18905	-8577.31	260	120
500	26.149542	0	18905	-8577.31	231	103

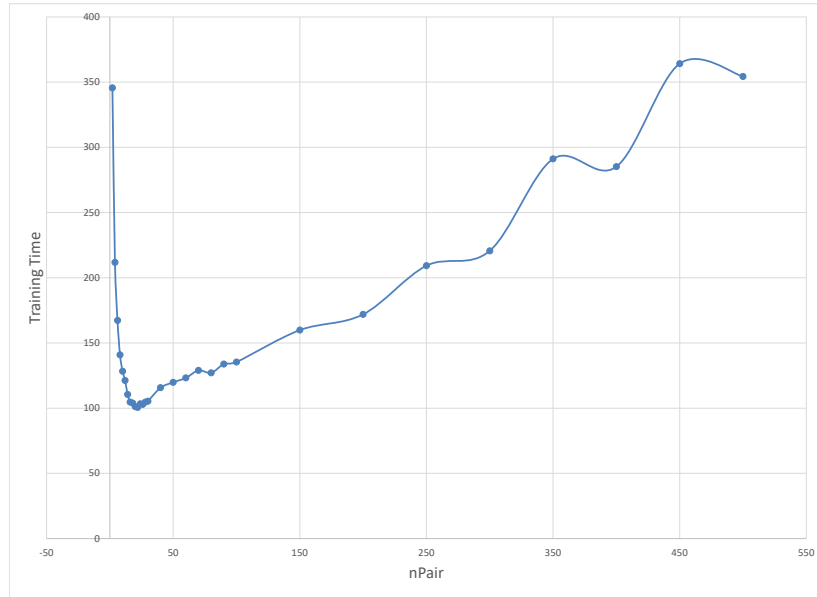


Figure 6.28: NRAL - Classification time for different pairs - Test 25.

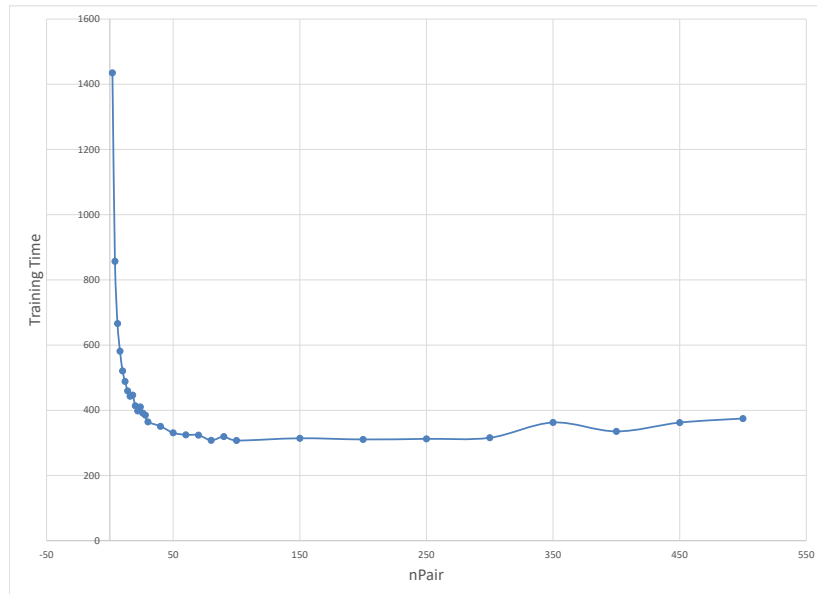


Figure 6.29: NRAL - Classification time for different pairs - Test 30.

Table 6.14: NRAL-Newton results for Test 25 with varying nPairs

<i>nPairs</i>	NRAL					
	<i>Time</i> (s)	<i>Err</i>	<i>nSv</i>	<i>Obj_{val}</i>	<i>Iterations</i>	<i>nD</i>
2	345.6106605	0	11138	-17599.1	171523	57713
4	211.7964885	0	11148	-17599	82153	31283
6	167.3644555	0	11190	-17598.8	64745	22417
8	140.941653	0	11191	-17597.1	59011	17575
10	128.3091945	0	11292	-17596.1	57314	14786
12	121.274775	0	11376	-17594.5	51603	12751
14	110.5670705	0	11208	-17590.2	41474	10935
16	104.652108	0	11187	-17588.4	38514	9672
18	103.92649	0	11181	-17583.7	37127	8819
20	101.1201545	0	11198	-17580.1	35153	8023
22	100.5937265	0	11166	-17580.6	32804	7374
24	103.4124165	0	11181	-17569.2	31746	6821
26	102.9410355	0	11192	-17568.3	30255	6399
28	104.704052	0	11179	-17560.3	28844	5942
30	105.373458	0	11186	-17560.3	27431	5608
40	115.6930665	0	11219	-17529.9	22966	4353
50	119.831452	0	11233	-17511.3	20387	3557
60	123.2320655	0	11265	-17485.3	17374	3035
70	128.9847075	0	11273	-17464.6	16217	2656
80	127.085432	0	11281	-17437.3	14667	2349
90	133.9062045	0	11302	-17417.6	13734	2122
100	135.398295	0.01	11317	-17383.8	12565	1938
150	159.8662785	0.01	11359	-17291.2	9286	1348
200	171.947201	0.03	11417	-17170.1	7691	1059
250	209.2557645	0.04	11492	-17062	6527	878
300	220.6572745	0.05	11488	-16935.4	5611	737
350	291.185642	0.06	11533	-16865.9	4948	652
400	285.2023365	0.07	11565	-16795.2	4543	579
450	364.198881	0.06	11621	-16699.6	4164	527
500	354.3726735	0.06	11645	-16645.5	3875	485

Table 6.15: NRAL-Newton results for Test 30 with varying nPairs

<i>nPairs</i>	NRAL					
	<i>Time</i> (s)	<i>Err</i>	<i>nSv</i>	<i>Obj_{val}</i>	<i>Iterations</i>	<i>nD</i>
2	1435.049495	0	96304	-49916.9	161068	70128
4	856.850389	0	96304	-49916.9	83233	34923
6	665.953668	0	96306	-49916.9	61243	23268
8	581.189752	0	96306	-49916.9	43342	17455
10	520.864897	0	96308	-49916.9	31937	14043
12	488.572578	0	96305	-49916.9	25423	11705
14	459.533196	0	96308	-49916.9	24948	9971
16	442.942795	0	96306	-49916.9	22643	8798
18	446.479111	0	96311	-49916.9	18954	7945
20	414.068777	0	96307	-49916.9	18279	7120
22	397.868199	0	96309	-49916.9	14246	6418
24	410.61023	0	96310	-49916.9	13844	6026
26	391.48879	0	96311	-49916.9	12682	5548
28	385.121373	0	96309	-49916.9	11326	5104
30	364.710254	0	96307	-49916.9	12115	4775
40	351.016469	0	96312	-49916.9	8332	3601
50	330.934029	0	96309	-49916.9	6462	2867
60	324.956495	0	96312	-49916.9	6117	2415
70	323.789219	0	96311	-49916.9	5285	2073
80	308.067251	0	96310	-49916.9	4455	1813
90	319.501155	0	96310	-49916.9	4089	1637
100	307.41373	0	96310	-49916.9	3835	1489
150	314.267004	0	96312	-49916.9	2805	990
200	310.617533	0	96313	-49916.9	2617	756
250	312.509337	0	96312	-49916.9	2179	606
300	315.741056	0	96312	-49916.9	1897	501
350	362.599668	0	96312	-49916.9	1735	443
400	335.358444	0	96311	-49916.9	1445	382
450	362.354545	0	96313	-49916.9	1433	344
500	374.846995	0	96313	-49916.9	1379	306

It is observed (Fig. 6.31), that a smaller number of pairs (10-50) performed best for NRAL. One reason for this is that finding the solution to the linear system in the subproblem that uses the newton solution takes longer to find with a larger number of pairs p values.

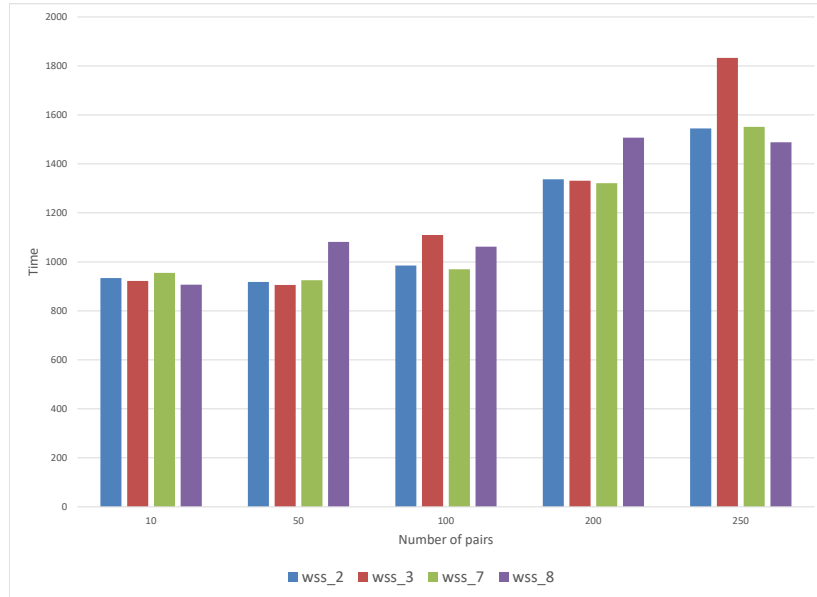


Figure 6.30: NRAL - Number of decompositions for different number of pairs.

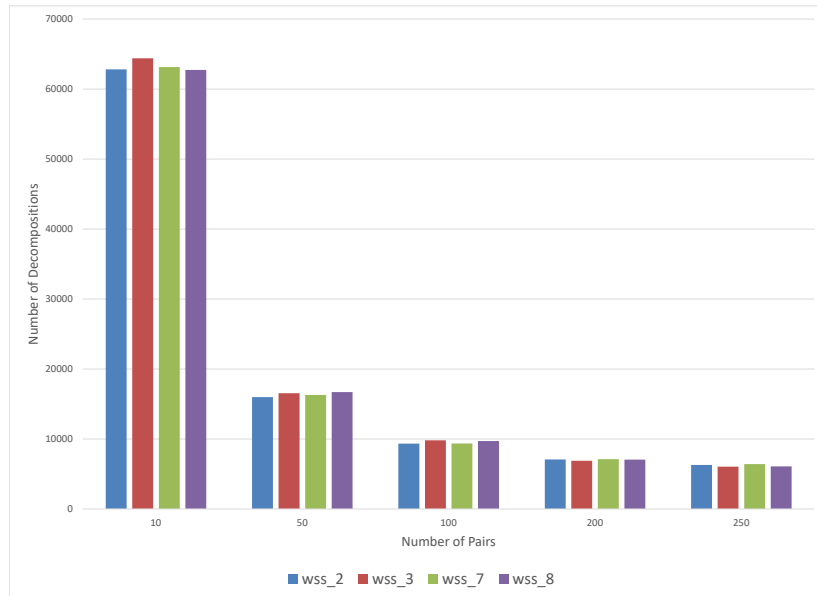


Figure 6.31: NRAL - Classification time for different working set selection schemes.

Comparison between LibSVM, NRAL and ALFPGM

We compared the performance of NRAL and ALFPGM with LibSVM implemented in `sklearn.svm.SVC` [24] that uses the SMO algorithm. All 3 methods computed the kernels on the fly and use a similar LRU kernel caching strategy. Tables 6.16 shows the comparison of results between LibSVM, NRAL, and ALFPGM using WSS_7 respectively. It has classification error (Err), training time ($time$), number of decompositions ($nDcmp$) and number of support vectors (nSv). Fig. 6.31 shows that the training time comparison between LibSVM, NRAL and ALFPGM for each of the 30 tests using WSS_8 . Fig. 6.32 shows the classification error comparison between LibSVM, NRAL and ALFPGM. Fig. 6.34 - 6.36 show the relative training times between LibSVM, NRAL and ALFPGM. The data points are arranged in the descending order of the number of training data m . Fig. 6.34 shows the ratios of training times between NRAL and ALFPGM. The values smaller than 1.0, shown in red indicate that the NRAL was faster, that is, it took more time for ALFPGM to train an SVM for a particular data set, while the values less than 1.0 shown in blue indicate that the ALFPGM was faster. Similarly, Fig. 6.35 shows the training time ratios between NRAL and LibSVM. Finally, Fig. 6.36 shows the training time ratios between ALFPGM and LibSVM.

Similarly to previous results, Fig. 6.33 shows that NRAL outperforms ALFPGM in training time to obtain a similar classification error. ALFPGM being a first-order method, it takes a while to converge in relation to the NRAL. In the few cases where ALFPGM outperformed NRAL, it was a fraction of a second, thus the performance gains were insignificant. Fig. 6.35 shows a comparison between NRAL and LibSVM for the same accuracy values. NRAL outperformed LibSVM especially in the cases where the number of data points were high. Analysis of a case where LibSVM outperformed NRAL, Test 27 shows that NRAL found more support vectors and achieved a lower classification error. In the other significant cases, in Tests 22 and 23, LibSVM outperformed NRAL. Overall, NRAL

outperformed LibSVM as a whole considering all 30 test cases. Fig. 6.37 shows the normalized training time, that is, training time divided by the number of training data sizes for all 3 methods. It clearly shows the training time reduction achieved by NRAL over LibSVM and ALFPGM.

Table 6.16: ALFPGM, NRAL, LibSVM (scikit-learn) comparison - WSS_7

			ALFPGM					LibSVM					NRAL							
Test	m	n	Err	Time (s)	nD	nSv	Obj _{val}	Err	Time (s)	nD	nSv	Obj _{val}	Err	Time (s)	nD	nSv	Obj _{val}			
01	100	10000	0	0.053843	12	100	-49.28	0	0.1615	296	100	-49.28	0	0.052339	11	100	-49.28			
02	625	4	0	0.039404	10	97	-359.79	0	0.0103	2239	97	-359.79	0	0.133891	49	98	-359.79			
03	1348	4	0	0.112512	20	418	-68.5117	0	0.0323	1316	412	-68.5116	0	0.394534	79	420	-68.5087			
04	569	30	0	0.105108	11	569	-262.212	0	0.0247	1068	569	-262.212	0	0.07197	26	569	-262.212			
05	1052	856	0	0.24199	15	1052	-199.872	0	0.9447	3109	1052	-199.872	0	0.231731	51	1052	-199.872			
06	1390	9	2.3	0.391218	30	1390	-7043.52	2.3	0.0889	2534	1390	-7043.52	2.3	0.312791	92	1390	-7043.52			
07	600	20000	0	0.201107	2	600	-300	0	4.3104	300	600	-300	0	0.250618	6	600	-300			
08	800	100000	0	9.431559	10	800	-140.787	0	61.7233	1970	800	-140.79	0	9.584948	31	800	-140.79			
09	14980	14	0	12.56972	198	14980	-7411.46	0	37.2218	47701	14980	-7411.46	0	11.08935	933	14980	-7411.46			
10	11500	178	0	10.24101	175	11500	-3679.4	0	136.5733	35819	11500	-3679.4	0	7.805595	716	11500	-3679.4			
11	13910	128	0	8.156131	168	4154	-914.7	0	11.6995	9575	4121	-919.284	0	6.736714	474	4188	-913.705			
12	289	3	2.08	0.052789	8	261	-1340.6	2.08	0.0065	645	261	-1340.6	2.08	0.115762	31	261	-1340.6			
13	1385	29	0	0.204953	16	1385	-691.624	0	0.1126	3443	1385	-691.623	0	0.177235	57	1385	-691.624			
14	4224	561	0	1.080599	23	4224	-2110.03	0	9.9981	10791	4224	-2110.77	0	1.287923	186	4224	-2110.77			
15	2600	500	0	0.29068	7	2600	-1300	0	3.1951	1300	2600	-1300	0	0.277331	26	2600	-1300			
16	19020	11	0	22.13577	294	18905	-8577.31	0	52.3462	62399	18905	-8577.31	0	20.14508	1240	18905	-8577.31			
17	5473	10	0.19	2.288623	87	5403	-2912.06	0.19	0.9113	11283	5403	-2912.06	0.19	1.455357	219	5403	-2819.44			
18	210	7	0	0.037389	9	95	-58.4927	0	0.0034	287	96	-58.4927	0	0.08296	25	100	-58.4927			
19	4435	36	0	1.149865	53	4435	-1505.53	0	1.1494	11628	4435	-1505.53	0	1.206499	228	4435	-1505.53			
20	7494	16	0	1.072134	21	7494	-3744.29	0	3.3169	16031	7494	-3746.73	0	2.0713	292	7494	-3746.73			
21	36974	123	4.59	100.483	843	36974	-347874	4.59	1215.305	110026	36974	-350686	4.59	115.5645	3439	36974	-347782			
22	26714	10	4.41	46.60157	485	26714	-231591	4.41	55.2686	39953	26714	-244605	4.41	42.49623	1934	26714	-233782			
23	45211	16	0	77.09719	508	45211	-9340.53	0	213.2611	96930	45211	-9340.53	0	84.99471	1987	45211	-9340.53			
24	18201	4295	0	139.096	275	18201	-5175.65	0	7870.151	49244	18201	-5175.65	0	124.434	1094	18201	-5175.65			
25	40768	10	0.2	81.7774	659	11701	-16033.5	0	166.716	79748	11123	-17599.1	0	113.7619	3351	11255	-17458.1			
26	32062	118	0.27	59.22488	630	20998	-15232.9	0.15	399.0199	49785	21016	-20231.4	0.31	58.60276	1752	21153	-13074.5			
27	45312	8	0.92	337.4581	2551	32681	-120386	0.7	336.2932	170816	32571	-135219	0.87	325.8839	8624	32788	-119744			
28	88588	6	0.05	359.1051	1223	43391	-25071.8	0.03	521.7201	122003	43331	-26753.4	0.05	329.9529	3868	43442	-24589.2			
29	98528	101	0.05	932.8615	1251	98411	-55816.7	0.03	5231.517	166893	98390	-56792.7	0.03	664.5909	3325	98393	-56647.4			
30	96320	21	0	133.8112	327	96302	-49869.2	0	1056.651	138749	96304	-49916.9	0	326.0516	2857	96304	-49916.9			
Total time				2337.372					17389.73								2249.817			

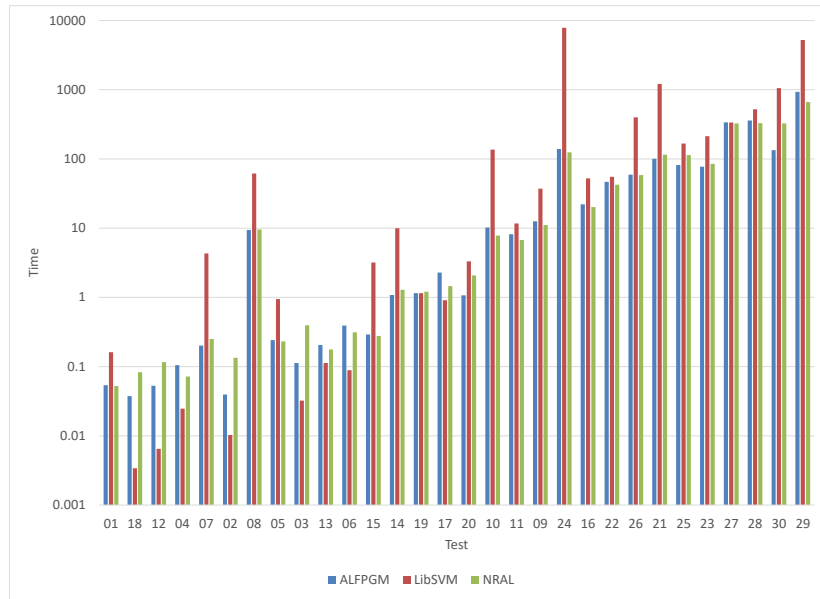


Figure 6.32: ALFPGM, NRAL and LibSVM (scikit-learn) timing comparison - WSS_7

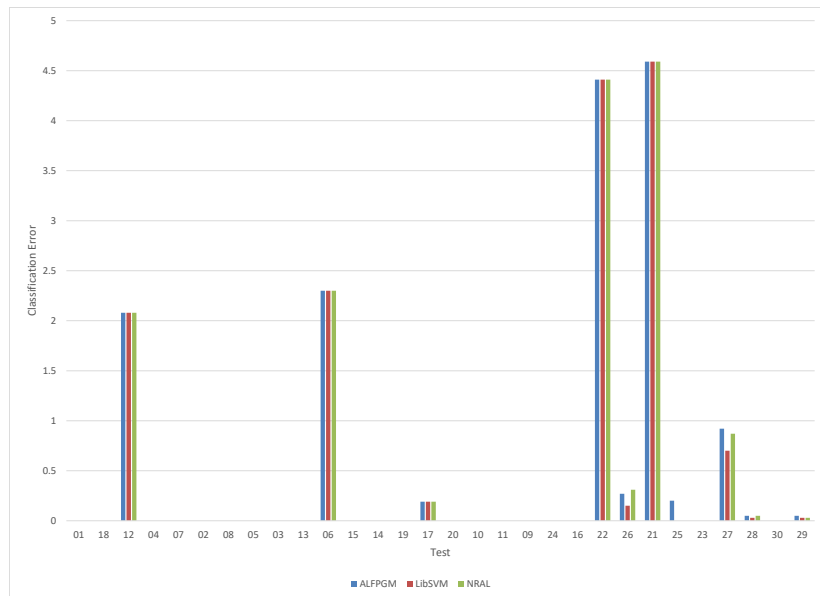


Figure 6.33: ALFPGM, NRAL, and LibSVM Classification Error Comparison - WSS_7

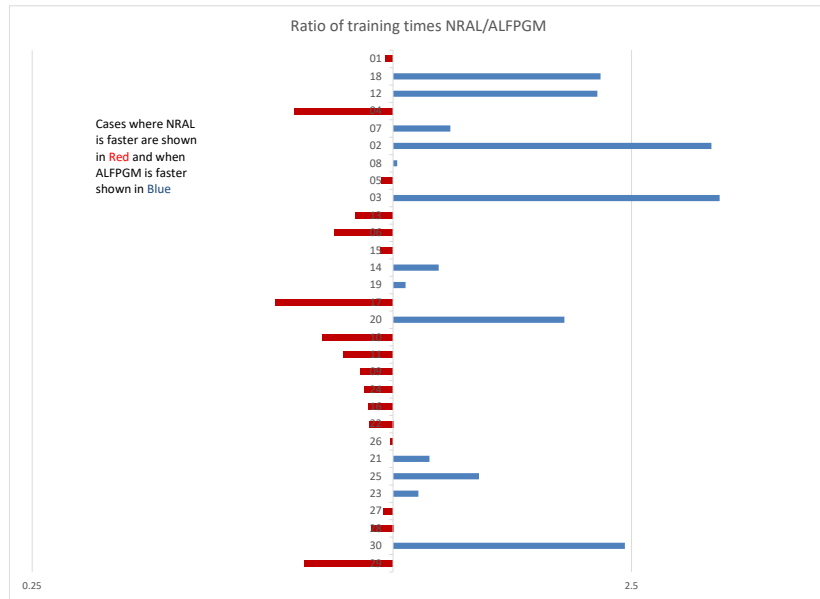


Figure 6.34: Comparison of NRAL and ALFPGM results - WSS_7

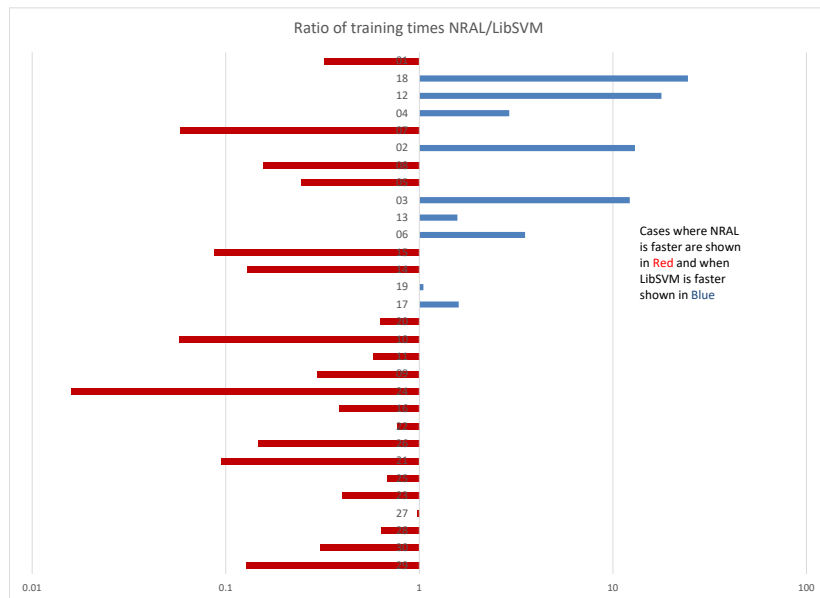


Figure 6.35: Comparison of the results of NRAL and LibSVM - WSS_7

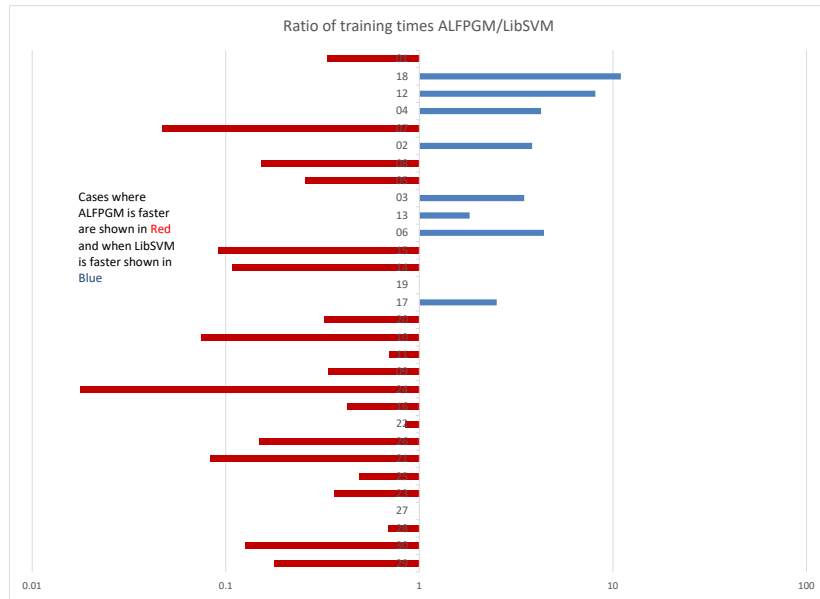


Figure 6.36: Comparison of the results of ALFPGM and LibSVM - WSS_7

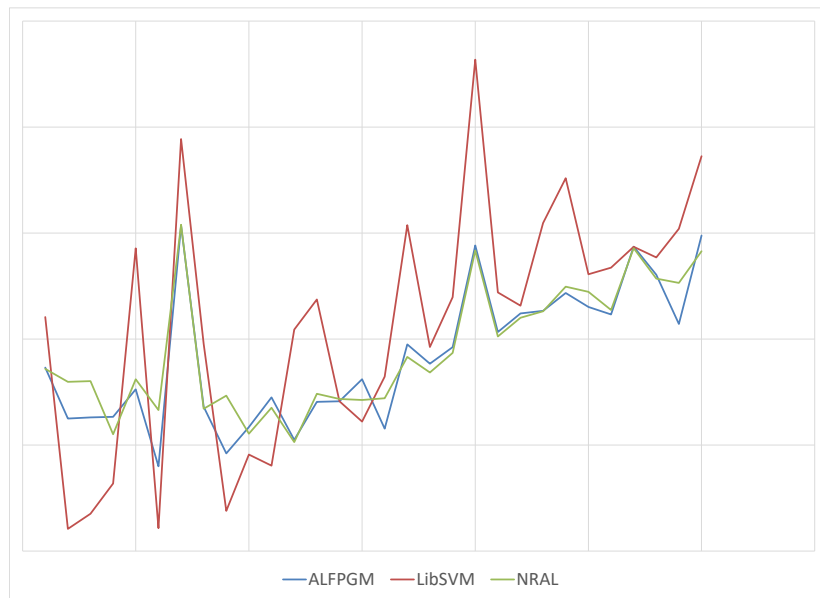


Figure 6.37: Comparison of normalized SVM training times - WSS_7

Table 6.17: Variation in data set size m and training time ratio with LibSVM

	Training time Ratio	
Test Size	LibSVM/ALFPGM	LibSVM/NRAL
$m < 10000$	0.383452	0.200755
$10000 < m < 40000$	3.048019	3.415291
$40000 < m < 80000$	1.616816	1.463441
$80000 < m$	3.020349	2.808884

In Table 6.17, the average normalized training time is calculated for different categories of size of the working set m . When $m < 10000$, LibSVM outperforms NRAL and ALFPGM with decomposition. An argument can be made that for such a small number of p , LibSVM is preferred or NRAL without decomposition should be used. Our focus is on training large data set problems, in which case according to the numerical results (Table 6.17), NRAL outperformed LibSVM in terms of training time for cases where m is large. Additionally, as mentioned above, in some of the cases where LibSVM gave a faster time, the LibSVM solution gave a higher classification error than the SVM solution obtained using NRAL.

We evaluated the testing error, classification error and training times using NRAL, ALFPGM and LibSVM for **ijcnn1** and **w8a** test and training data from LibSVM dataset. The results are shown in Table 6.18.

Table 6.18: Classification and testing error **ijcnn1** using ALFPGM, NRAL and LibSVM

Test	method	$nWss$	Err	test_err	total_seconds	Obj_{val}
	LibSVM		0	4.13	165.05485	-5464.916478
ijcnn1	NRAL	7	0	4.11	81.77327	-5456.129607
ijcnn1	ALFPGM	7	0	4.13	83.67837	-5463.847787
49,990 training data (45137 +1, 4853 -1). 22 features in data points.						
91,701 testing data (8712 +1, 82989 -1).						
https://www.csie.ntu.edu.tw/~cjlin/LibSVMtools/datasets/binary.html						

Table 6.19: Classification and testing error **w8a** using ALFPGM, NRAL and LibSVM

Test	method	nW_{ss}	Err	test_err	total_seconds	Obj_{val}
	LibSVM		0.83	0.91	143.4498	-100856.712082
w8a	NRAL	7	0.84	0.93	65.8715	-98818.20693
w8a	ALFPGM	7	0.88	0.95	93.3837	-66389.65889
49,749 training data (1479 +1, 48270 -1). 22 features in data points. 14,951 testing data (454 +1, 14497 -1). https://www.csie.ntu.edu.tw/~cjlin/LibSVMtools/datasets/binary.html						

We tested the performance of NRAL and LibSVM for a very large dataset. We selected binary classification data from <https://www.openml.org/>; **Agrawal1** with 1,000,000 instances and 10 features. LibSVM took approximately 13.4 hours to train the data, while NRAL with $p = 40$ took approximately 8 hours to train the data. This is a significant reduction in training time.

Table 6.20: NRAL (40 Pairs) vs LibSVM for very large dataset

	m	1000000				
	n	9				
	Err	Time	Obj_{val}	nD	nSv	
LibSVM	0.08	48402.69684	-1105728.522271	2735718	180447	
NRAL (40 Pairs)	0.08	28942.82873	-892233.474715	36088	183222	

Summary

In this chapter, we have presented numerical results. We have presented and discussed the results from using the different working set selection schemes. We have evaluated the performance of ALFPGM using different project gradient methods and presented numerical results. The numerical results show that MFISTA generally gives the least training time and

classification error. We have shown the relative performance of NRAL and ALFPGM versus the widely used LibSVM (`sklearn.svm.SVC`). Numerical results for NRAL and ALFPGM applied to the SVM indicate that ALFPGM and NRAL are feasible training algorithms for SVMs. In many of the cases presented, especially when training large dataset problems, NRAL and ALFPGM trained the SVM faster than LibSVM with similar classification error, training time and objective value.

Chapter 7: Concluding Remarks

In this thesis, we investigated the question of efficiently training of Support Vector Machines (SVM) using Lagrangian multipliers methods. We studied the Nonlinear Rescaling Augmented Lagrangian (NRAL) and the Augmented Lagrangian Fast Projected Gradient Method (ALFPGM) methods for Support Vector Machine training, two Lagrangian multipliers methods. We improved on previous work that utilized some form of ALFPGM and NRAL and adapted them for support vector machine training using high performance computing techniques and SVM decomposition techniques. We proposed better working set selection schemes to define the SVM decomposition sub-problem. Highly optimized linear algebra routines were used in the implementation to speed up some of the computations. Numerical results indicate that the developed NRAL solver is competitive with the widely used SVM solver, LibSVM. In many cases especially for the large SVM problems, NRAL outperformed LibSVM achieving faster training times while having similar or lower classification error rates.

To train large SVM problems, it is our recommendation to use NRAL over LibSVM. Using pairs p in the working set reduced the number of decomposition iterations needed to solve the SVM problem. By using an efficient solver like NRAL, we have been able to outperform LibSVM especially for those cases with large training data sizes with training data greater than 80,000.

We proposed different working set selection schemes for p pair decomposition problems. Working set selection takes a large portion of the training time thus it is important to optimize working set selection. Numerical results for the proposed working set selection methods showed training time was reduced for all problems solved in comparison to working set methods established in the literature. We have shown that in some examples α selection

can be truncated early without a compromise in the overall results and this leads to a significant reduction in training time. The WSS selected is important for the performance of the SVM training solution. It is our recommendation to use the proposed WSS_7 and WSS_8 in working set selection where $p > 2$ pairs are selected. This can be used with ALFPGM and NRAL or any other non dual decomposition SVM training algorithms that require a working set size of more than 2. Numerical results indicates that the optimal choice of the number of pairs in the working set p for NRAL is 10 - 50 pairs while the recommended number of pairs for ALFPGM is 100-250 pairs.

We evaluated the performance of ALFPGM using different project gradient methods. Numerical results show that Greedy FISTA gave faster training time while achieving the same or lower classification error. Monotone FISTA (MFISTA) requires the computation of the minimized function thus requiring more resources and it gave the longest training time and there is no advantage to using it. It is recommended based on the numerical results not to use MFISTA for the inner fast projected optimization in the ALFPGM technique. RADA and Restart FISTA also gave comparable training times as Greedy FISTA which had a slight edge over them in terms of overall training time.

NRAL showed a 2x faster training time than LibSVM that has been modified to use parallel processing for very large training dataset. ALFPGM shows promising results in finding the SVM sub problem solution. It is slower than LibSVM and NRAL. However, it is a simpler algorithm to implement. ALFPGM needed more inner iterations to find the solution to SVM sub problem in comparison to NRAL. If the computations can be made faster, ALFPGM shows promising outlook for solving training SVM. A large part of the training time goes in to working set selection. To further reduce the training time, the focus of optimizing the SVM training problem should be the working set selection.

Future work

The results of this thesis provide a strong foundation for future work in SVM training using Lagrange multiplier methods with decomposition using multiple pairs working selection. It is our take that if the WSS can be further improved for multiple pair selection, the total training time will be significantly reduced. This can be done by efficiently computing kernel values and using a different cache scheme to store previously computed kernels.

The choice of the number of pairs p is important as it determines the number of decompositions. Our work has indicated a range of values based on numerical results for the optimal value of p . A future work area will be to optimally determine the number of pairs p based on the size of the problem.

Selecting optimal values of $WSS_{minAlphaCheck}$ and $minAlphaOpt$ used in WSS_3 , WSS_4 , WSS_7 , WSS_8 needs to be further explored. Finding the optimal values for this parameters based on the input data are still open questions.

Graphics processing units are getting cheaper and they are also incorporating larger memory and faster clock speed. The use of GPU should be revisited in future to consider the possibility of computing the kernel matrix on the GPU. The ALFPGM and NRAL algorithms can also be implemented on the GPU. ALFPGM will particularly benefit from using GPU as the mathematical operations are simple matrix operations.

Summary of research contributions

This thesis has contributed to the field of supervised machine learning by exploring the use of decomposition method and using Lagrange multiplier algorithms to solve the SVM subproblem. We have revisited the SVM problem that is currently mostly solved using dual decomposition methods. We have presented new faster working set selection algorithms that have been demonstrated to reduce the SVM training time.

In summary, the work described in this thesis made the following contributions:

- Developed an improved ALFPGM algorithm with SVM Decomposition solution. We

have developed and implemented ALFPGM with high-performance computing techniques to reduce training time. We compared different FISTA algorithms in ALFPGM for SVM training. To the best of our knowledge, it is a first attempt to use ALFPGM with decomposition for SVM training. So far FISTA has been used only for training small datasets whose kernel matrices can fully fit in the computer memory.

- Developed Nonlinear Rescaling Augmented Lagrangian (NRAL) with newton for SVM Decomposition solution. We have developed and implemented NRAL with high performance computing techniques to reduce training time. Previous works like [50] have studied Nonlinear Rescaling with SVM . The difference with our work is that we have applied the decomposition techniques and have expanded the area by studying how the algorithm performs with different decomposition methods. Numerical results shows that the NRAL method developed trained large set SVM problems faster than the widely used LibSVM that is used for training SVM.
- Proposed and developed working set selection schemes ($WSS_3, WSS_4, WSS_7, WSS_8$) for selection of p number of pairs where $p > 2$ to reduce SVM training times. Previously, **m**aximum **v**iolating **p**air (MVP) and working set selection using 2nd order statistics have been used for working set selection. Some works like [27] have used $p > 2$ pairs in the working set selection. We recommend four new working set selection schemes to be used for training non dual working set SVM training algorithms.
- Finally, we studied how different parameters selections affect the SVM training time, as well as the classification errors. We examined how the choice of pairs p affects the overall training results and gave recommendations on choosing the parameters.

Conferences and Publication

We have published and presented the following works in the course of working on this thesis.

- M. Aregbesola and I. Griva, Augmented Lagrangian fast projected gradient algorithm

with working set selection for training support vector machines, J. Appl. Numer. Optim, vol. 3, no. 1, pp. 320, 2021. [Online]. Available:<http://jano.biemdas.com/archives/1224> [91].

- Training large data set SVM with Augmented Lagrangian - Fast Projected gradient method, 2021 INFORMS Annual Meeting, October 2021
- Numerical Experiments with training support vector machine on midsized data sets with Lagrange multipliers methods, 2020 INFORMS Annual Meeting, November 2020

Appendix A: Appendix

A.1 SVM and Other classifiers

A comparison of a several classifiers in **Scikit-learn** were done using **w3a** and **w8a**. SVM shows a high generalization accuracy, i.e. lower testing data miss-classification error.

Table A.1: scikit-learn run for tests w8a and w3a

	Method	Classification Error	Classification Error	Training Time	Testing Time (s)
w8a	Nearest Neighbors	98.73	98.64	6.1426	1041.2021
	RBF SVM	99.17	99.09	92.0694	133.3093
	Decision Tree	97.67	97.52	0.3225	0.3828
	Random Forest	97.03	96.96	0.1057	0.1911
	Neural Net sgd	98.45	98.37	213.4776	213.8627
	AdaBoost	98.2	98.23	6.9574	9.4515
	Naive Bayes	76.1	75.95	0.253	0.5879
	QDA	48.75	49.06	1.252	2.0009
w3a	Nearest Neighbors	98.57	97.32	0.1559	86.5544
	RBF SVM	99.25	98.27	0.9608	7.2794
	Decision Tree	98.13	97.62	0.0272	0.0773
	Random Forest	97.21	97.12	0.0299	0.1032
	Neural Net sgd	97.09	97.02	21.3974	21.7364
	AdaBoost	98.51	97.91	0.5837	2.6098
	Naive Bayes	95.13	94.29	0.0211	0.2778
	QDA	98.6	97.97	0.4131	1.0216

A.2 Source Code

The source code for this thesis is available at <https://github.com/optimcode/svmsolvers>

Bibliography

- [1] H. Yu and S. Kim, *SVM Tutorial — Classification, Regression and Ranking*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 479–506. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-92910-9_15
- [2] V. Vapnik and A. Chervonenkis, *Theory of Pattern Recognition [in Russian]*. Moscow: Nauka, 1974, (German Translation: W. Wapnik & A. Tscherwonienkis, *Theorie der Zeichenerkennung*, Akademie-Verlag, Berlin, 1979).
- [3] V. N. Vapnik, *The Nature of Statistical Learning Theory*. New York, NY, USA: Springer-Verlag New York, Inc., 1995.
- [4] C. J. Burges, “A tutorial on support vector machines for pattern recognition,” *Data mining and knowledge discovery*, vol. 2, no. 2, pp. 121–167, 1998.
- [5] A. J. Smola and B. Schölkopf, “A tutorial on support vector regression,” *Statistics and Computing*, vol. 14, no. 3, pp. 199–222, Aug. 2004. [Online]. Available: <http://dx.doi.org/10.1023/B:STCO.0000035301.49549.88>
- [6] R. Herbrich, T. Graepel, and K. Obermayer, “Large margin rank boundaries for ordinal regression,” in *Advances in Large Margin Classifiers*, P. J. Bartlett, B. Schölkopf, D. Schuurmans, and A. J. Smola, Eds. MIT Press, 2000, pp. 115–132.
- [7] J. H. Friedman, “Another approach to polychotomous classification,” Department of Statistics, Stanford University, Tech. Rep., 1996. [Online]. Available: <http://www-stat.stanford.edu/~jhf/ftp/poly.ps.Z>
- [8] T. Hastie and R. Tibshirani, “Classification by pairwise coupling,” in *Proceedings of the 1997 Conference on Advances in Neural Information Processing Systems 10*, ser. NIPS ’97. Cambridge, MA, USA: MIT Press, 1998, pp. 507–513. [Online]. Available: <http://dl.acm.org/citation.cfm?id=302528.302744>
- [9] L. Bottou and C. Lin, “Support vector machine solvers,” in *Large Scale Kernel Machines*, L. Bottou, O. Chapelle, D. DeCoste, and J. Weston, Eds. Cambridge, MA.: MIT Press, 2007, pp. 301–320. [Online]. Available: <http://leon.bottou.org/papers/bottou-lin-2006>
- [10] M. C. Ferris and T. S. Munson, “Interior-point methods for massive support vector machines,” *SIAM Journal on Optimization*, vol. 13, no. 3, pp. 783–804, 2002.

- [11] R. J. Vanderbei and D. F. Shanno, "An interior-point algorithm for nonconvex nonlinear programming," *Comput. Optim. Appl.*, vol. 13, no. 1-3, pp. 231–252, Apr. 1999. [Online]. Available: <http://dx.doi.org/10.1023/A:1008677427361>
- [12] R. Polyak and M. Teboulle, "Nonlinear rescaling and proximal-like methods in convex optimization," *Mathematical Programming*, vol. 76, no. 2, pp. 265–284, Feb. 1997. [Online]. Available: <https://doi.org/10.1007/BF02614440>
- [13] I. Griva and R. A. Polyak, "1.5-q-superlinear convergence of an exterior-point method for constrained optimization," *Journal of Global Optimization*, vol. 40, no. 4, pp. 679–695, Apr 2008. [Online]. Available: <https://doi.org/10.1007/s10898-006-9117-x>
- [14] V. Bloom, I. Griva, B. Kwon, and A. R. Wolff, "Exterior-point method for support vector machines," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 25, no. 7, pp. 1390–1393, July 2014.
- [15] E. Osuna, R. Freund, and F. Girosi, "An improved training algorithm for support vector machines," in *Neural Networks for Signal Processing [1997] VII. Proceedings of the 1997 IEEE Workshop*, 1997, Conference Proceedings, pp. 276–285.
- [16] C. Saunders, M. O. Stitson, J. Weston, L. Bottou, A. Smola *et al.*, "Support vector machine-reference manual," Royal Holloway, University of London, Technical Report CSD-TR-98-03, 1998.
- [17] T. Joachims, *Making large-scale support vector machine learning practical*. MIT Press, 1999, book section Solving the Quadratic Programming Problem Arising in Support Vector Classification, pp. 169–184. [Online]. Available: <http://dl.acm.org/citation.cfm?id=299094.299103>
- [18] S. Keerthi and E. Gilbert, "Convergence of a generalized smo algorithm for svm classifier design," *Machine Learning*, vol. 46, no. 1, pp. 351–360, Jan 2002. [Online]. Available: <https://doi.org/10.1023/A:1012431217818>
- [19] J. Platt, "Sequential minimal optimization: A fast algorithm for training support vector machines," Microsoft Corporation, Tech. Rep., Apr. 1998. [Online]. Available: <https://www.microsoft.com/en-us/research/publication/sequential-minimal-optimization-a-fast-algorithm-for-training-support-vector-machines/>
- [20] N. Cristianini and J. Shawe-Taylor, *An Introduction to Support Vector Machines: And Other Kernel-based Learning Methods*. New York, NY, USA: Cambridge University Press, 2000.
- [21] S. S. Keerthi, S. K. Shevade, C. Bhattacharyya, and K. R. K. Murthy, "Improvements to platt's smo algorithm for svm classifier design," *Neural Comput.*, vol. 13, no. 3, pp. 637–649, Mar. 2001. [Online]. Available: <http://dx.doi.org/10.1162/089976601300014493>
- [22] C.-C. Chang and C. Lin, "Libsvm: A library for support vector machines," *ACM Trans. Intell. Syst. Technol.*, vol. 2, no. 3, pp. 27:1–27:27, May 2011, software available at <http://www.csie.ntu.edu.tw/~cjlin/libsvm>. [Online]. Available: <http://doi.acm.org/10.1145/1961189.1961199>

- [23] R.-E. Fan, P.-H. Chen, and C.-J. Lin, “Working set selection using second order information for training support vector machines,” *J. Mach. Learn. Res.*, vol. 6, pp. 1889–1918, Dec. 2005. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1046920.1194907>
- [24] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, “Scikit-learn: Machine learning in Python,” *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.
- [25] P. Chang, Z. Bi, and Y. Feng, “Parallel smo algorithm implementation based on openmp,” in *2014 IEEE International Conference on System Science and Engineering (ICSSE)*, July 2014, pp. 236–240.
- [26] L. J. Cao, S. S. Keerthi, C.-J. Ong, J. Q. Zhang, U. Periyathamby, X. J. Fu, and H. P. Lee, “Parallel sequential minimal optimization for the training of support vector machines,” *Trans. Neur. Netw.*, vol. 17, no. 4, pp. 1039–1049, Jul. 2006. [Online]. Available: <http://dx.doi.org/10.1109/TNN.2006.875989>
- [27] W. Wei, C. Li, and J. Guo, “Improved parallel algorithms for sequential minimal optimization of classification problems,” in *2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, 2018, pp. 6–13.
- [28] A. Athanasopoulos, A. Dimou, V. Mezaris, and I. Kompatsiaris, “Gpu acceleration for support vector machines,” in *Proc. 12th International Workshop on Image Analysis for Multimedia Interactive Services (WIAMIS 2011)*, Delft, The Netherlands,, 04 2011.
- [29] A. Carpenter, “Cusvm: A cuda implementation of support vector classification and regression. <http://patternsonascreen.net/cusvm.html>,” 2009.
- [30] B. Catanzaro, N. Sundaram, and K. Keutzer, “Fast support vector machine training and classification on graphics processors,” pp. 104–111, 2008.
- [31] A. Cotter, N. Srebro, and J. Keshet, “A gpu-tailored approach for training kernelized svms,” in *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD ’11. New York, NY, USA: ACM, 2011, pp. 805–813. [Online]. Available: <http://doi.acm.org/10.1145/2020408.2020548>
- [32] F. Sha, Y. Lin, L. K. Saul, and D. D. Lee, “Multiplicative updates for nonnegative quadratic programming,” *Neural Comput.*, vol. 19, no. 8, pp. 2004–2031, Aug. 2007. [Online]. Available: <http://dx.doi.org/10.1162/neco.2007.19.8.2004>
- [33] O. Chapelle, “Training a support vector machine in the primal,” *Neural Comput.*, vol. 19, no. 5, pp. 1155–1178, May 2007. [Online]. Available: <http://dx.doi.org/10.1162/neco.2007.19.5.1155>
- [34] J.-X. Dong, A. Krzyżak, and C. Y. Suen, “A fast parallel optimization for training support vector machine,” in *Proceedings of the 3rd International*

- Conference on Machine Learning and Data Mining in Pattern Recognition*, ser. MLDM'03. Berlin, Heidelberg: Springer-Verlag, 2003, pp. 96–105. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1759548.1759561>
- [35] T. Eitrich and B. Lang, *Data Mining with Parallel Support Vector Machines for Classification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 197–206. [Online]. Available: https://doi.org/10.1007/11890393_21
 - [36] —, “Efficient implementation of serial and parallel support vector machine training with a multi-parameter kernel for large-scale data mining,” *International Journal of Computer, Electrical, Automation, Control and Information Engineering*, vol. 1, no. 11, pp. 3585 – 3590, 2007. [Online]. Available: <http://waset.org/Publications?p=11>
 - [37] D. Brugger, “Parallel support vector machines,” in *In Proceedings of the IFIP International Conference on Very Large Scale Integration of System on Chip (VLSI-SoC)*. Springer, 10 2007.
 - [38] L. Zanni, T. Serafini, and G. Zanghirati, “Parallel software for training large scale support vector machines on multiprocessor systems,” *J. Mach. Learn. Res.*, vol. 7, pp. 1467–1492, Dec. 2006. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1248547.1248601>
 - [39] H.-X. Zhao and F. Magoulès, “Parallel support vector machines on multi-core and multiprocessor systems,” in *Proceedings of the 11th International Conference on Artificial Intelligence and Applications (AIA 2011)*, 2011.
 - [40] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa, “Natural language processing (almost) from scratch,” *Journal of Machine Learning Research*, vol. 12, pp. 2493–2537, Aug 2011. [Online]. Available: <http://leon.bottou.org/papers/collobert-2011>
 - [41] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C. Lin, “Liblinear: A library for large linear classification,” *J. Mach. Learn. Res.*, vol. 9, pp. 1871–1874, Jun. 2008. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1390681.1442794>
 - [42] T. Hazan, A. Man, and A. Shashua, “A parallel decomposition solver for svm: Distributed dual ascend using fenchel duality,” in *2008 IEEE Conference on Computer Vision and Pattern Recognition*, June 2008, pp. 1–8.
 - [43] J.-P. Zhang, Z.-W. Li, and J. Yang, “A parallel svm training algorithm on large-scale classification problems,” in *2005 International Conference on Machine Learning and Cybernetics*, vol. 3, Aug 2005, pp. 1637–1641 Vol. 3.
 - [44] Y. Nesterov, *Introductory lectures on convex optimization: a basic course*. Kluwer Academic Publishers, 2004.
 - [45] A. Beck and M. Teboulle, “A fast iterative shrinkage-thresholding algorithm for linear inverse problems,” *SIAM Journal on Imaging Sciences*, vol. 2, no. 1, pp. 183–202, 2009. [Online]. Available: <https://doi.org/10.1137/080716542>

- [46] —, “Fast gradient-based algorithms for constrained total variation image denoising and deblurring problems,” *IEEE Transactions on Image Processing*, vol. 18, no. 11, pp. 2419–2434, 2009.
- [47] R. A. Polyak, J. Costa, and S. Neyshabouri, “Dual fast projected gradient method for quadratic programming,” *Optimization Letters*, vol. 7, no. 4, pp. 631–645, Apr. 2013. [Online]. Available: <https://doi.org/10.1007/s11590-012-0476-6>
- [48] R. A. Polyak, “Projected gradient method for non-negative least square.” in *Infinite products of operators and their applications. A research workshop of the Israel Science Foundation, Haifa, Israel, May 21–24, 2012*. Providence, RI: American Mathematical Society (AMS), 2015, pp. 167–179.
- [49] I. Griva, “Convergence analysis of augmented lagrangian - fast projected gradient method for convex quadratic problems,” *Pure and Applied Functional Analysis*, pp. 417–428, 2018.
- [50] R. Polyak, S.-S. Ho, and I. Griva, “Support vector machine via nonlinear rescaling method,” *Optimization Letters*, vol. 1, no. 4, pp. 367–378, 2007.
- [51] C. Campbell and Y. Ying, “Learning with support vector machines,” *Synthesis Lectures on Artificial Intelligence and Machine Learning*, vol. 5, no. 1, pp. 1–95, 2011. [Online]. Available: <https://doi.org/10.2200/S00324ED1V01Y201102AIM010>
- [52] C. Han, M. Li, T. Zhao, and T. Guo, “An accelerated proximal gradient algorithm for singly linearly constrained quadratic programs with box constraints,” *TheScientific-WorldJournal*, vol. 2013, p. 246596, 01 2013.
- [53] V. Vapnik and A. Lerner, “Pattern Recognition using Generalized Portrait Method,” *Automation and Remote Control*, vol. 24, 1963.
- [54] I. Griva, S. Nash, and A. Sofer, *Linear and Nonlinear Optimization: Second Edition*. Society for Industrial and Applied Mathematics (SIAM, 3600 Market Street, Floor 6, Philadelphia, PA 19104), 2009. [Online]. Available: <http://books.google.com/books?id=uOJ-Vg1BnKgC>
- [55] D. Bertsekas, *Nonlinear Programming*. Athena Scientific, 1995.
- [56] Y. Lee and O. L. Mangasarian, “Rsvm: Reduced support vector machines,” in *Data Mining Institute, Computer Sciences Department, University of Wisconsin*, 2001, pp. 00–07.
- [57] S. Shalev-Shwartz, Y. Singer, N. Srebro, and A. Cotter, “Pegasos: primal estimated sub-gradient solver for svm,” *Mathematical Programming*, vol. 127, no. 1, pp. 3–30, Mar. 2011. [Online]. Available: <https://doi.org/10.1007/s10107-010-0420-4>
- [58] W. Karush, “Minima of functions of several variables with inequalities as side constraints,” Master’s thesis, Department of Mathematics, University of Chicago, 1939.

- [59] H. W. Kuhn and A. W. Tucker, “Nonlinear programming,” in *Proceedings of the Second Berkeley Symposium on Mathematical Statistics and Probability*, J. Neyman, Ed. Berkeley: University of California Press, 1951, pp. 481–492.
- [60] S. Boyd and L. Vandenberghe, *Convex Optimization*. New York, NY, USA: Cambridge University Press, 2004.
- [61] J. Nocedal, A. Wächter, and R. A. Waltz, “Adaptive barrier update strategies for nonlinear interior methods,” *SIAM Journal on Optimization*, vol. 19, no. 4, pp. 1674–1693, 2009. [Online]. Available: <https://doi.org/10.1137/060649513>
- [62] M. R. Hestenes, “Multiplier and gradient methods,” *Journal of Optimization Theory and Applications*, vol. 4, no. 5, pp. 303–320, Nov. 1969. [Online]. Available: <https://doi.org/10.1007/BF00927673>
- [63] P. M. J. D., “A method for nonlinear constraints in minimization problems,” *Optimization*, pp. 283–298, 1969.
- [64] R. Courant, “Variational methods for the solution of problems of equilibrium and vibrations,” *BULL. AMER. MATH. SOC*, 1943.
- [65] D. Bertsekas, *Constrained Optimization and Lagrange Multiplier Methods*. Athena Scientific, 1996.
- [66] A. R. Conn, N. I. M. Gould, and P. L. Toint, *Lancelot: A FORTRAN Package for Large-Scale Nonlinear Optimization (Release A)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1992.
- [67] J. Liang, T. Luo, and C.-B. Schnlieb, “Improving ”fast iterative shrinkage-thresholding algorithm”: Faster, smarter and greedier,” 2021.
- [68] P. Combettes and J.-C. Pesquet, “Proximal splitting methods in signal processing,” in *Fixed-Point Algorithms for Inverse Problems in Science and Engineering*, ser. Springer Optimization and Its Applications, H. H. Bauschke, R. S. Burachik, P. L. Combettes, V. Elser, D. R. Luke, and H. Wolkowicz, Eds. Springer New York, 2011, vol. 49, pp. 185–212. [Online]. Available: http://dx.doi.org/10.1007/978-1-4419-9569-8_10
- [69] Y. Nesterov, “A method for solving convex programming problem with convergence rate,” *Soviet Math. Dokl.*, vol. 27, pp. 372–376, 1983.
- [70] B. O’Donoghue and E. Candes, “Adaptive restart for accelerated gradient schemes,” 2012.
- [71] R. Polyak, “Modified barrier functions (theory and methods),” *Mathematical Programming*, vol. 54, no. 1, pp. 177–222, Feb. 1992. [Online]. Available: <https://doi.org/10.1007/BF01586050>
- [72] I. Griva and R. A. Polyak, “Primal-dual nonlinear rescaling method with dynamic scaling parameter update,” *Mathematical Programming*, vol. 106, no. 2, pp. 237–259, Apr 2006. [Online]. Available: <https://doi.org/10.1007/s10107-005-0603-6>

- [73] V. Bloom, “Exterior-point algorithms for solving large-scale nonlinear optimization problems,” Ph.D. dissertation, George Mason University, 2014.
- [74] Q. Yang, C. Li, and J. Guo, “Multi-order information for working set selection of sequential minimal optimization,” in *Proceedings of the Twenty-Second International Conference on Artificial Intelligence and Statistics*, ser. Proceedings of Machine Learning Research, K. Chaudhuri and M. Sugiyama, Eds., vol. 89. PMLR, 16–18 Apr 2019, pp. 3264–3272. [Online]. Available: <https://proceedings.mlr.press/v89/yang19b.html>
- [75] P.-H. Chen, R.-E. Fan, and C.-J. Lin, “A study on smo-type decomposition methods for support vector machines,” *Trans. Neur. Netw.*, vol. 17, no. 4, p. 893908, Jul. 2006. [Online]. Available: <https://doi.org/10.1109/TNN.2006.875973>
- [76] Q. Zhang, J. Wang, A. Lu, S. Wang, and J. Ma, “An improved smo algorithm for financial credit risk assessment evidence from chinas banking,” *Neurocomputing*, vol. 272, pp. 314 – 325, 2018. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0925231217312328>
- [77] C.-W. Hsu and C.-J. Lin, “A simple decomposition method for support vector machines,” *Mach. Learn.*, vol. 46, no. 1-3, pp. 291–314, Mar. 2002. [Online]. Available: <http://dx.doi.org/10.1023/A:1012427100071>
- [78] P. Pacheco, *An Introduction to Parallel Programming*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.
- [79] S. Tyree, J. R. Gardner, K. Q. Weinberger, K. Agrawal, and J. Tran, “Parallel support vector machines in practice,” *CoRR*, vol. abs/1404.1066, 2014. [Online]. Available: <http://arxiv.org/abs/1404.1066>
- [80] T. M. John Cheng, Max Grossman, *Professional CUDA C Programming*, 1st ed. Birmingham, UK, UK: Wrox Press Ltd., 2014.
- [81] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov, “Numerical linear algebra on emerging architectures: The plasma and magma projects,” *Journal of Physics: Conference Series*, vol. 180, no. 1, p. 012037, 2009. [Online]. Available: <http://stacks.iop.org/1742-6596/180/i=1/a=012037>
- [82] Intel Corporation, “Intel Math Kernel Library (Intel MKL),” <http://software.intel.com/en-us/intel-mkl>.
- [83] NVIDIA, “Nvidia cuda cublas library,” <https://developer.nvidia.com/cublas>.
- [84] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, T. Munson, K. Rupp, B. F. Smith, S. Zampini, H. Zhang, and H. Zhang, “PETSc users manual,” Argonne National Laboratory, Tech. Rep. ANL-95/11 - Revision 3.7, 2016. [Online]. Available: <http://www.mcs.anl.gov/petsc>

- [85] C. Lin, “On the convergence of the decomposition method for support vector machines,” *IEEE Transactions on Neural Networks*, vol. 12, no. 6, pp. 1288–1298, Nov 2001.
- [86] V. Bloom, I. Griva, and F. Quijada, “Fast projected gradient method for support vector machines,” *Optimization and Engineering*, vol. 17, no. 4, pp. 651–662, Dec. 2016. [Online]. Available: <https://doi.org/10.1007/s11081-016-9328-z>
- [87] D. Dua and C. Graff, “UCI machine learning repository,” 2017. [Online]. Available: <http://archive.ics.uci.edu/ml>
- [88] A. K. Menon, “Large-scale support vector machines: Algorithms and theory,” University of California, Tech. Rep., 2009.
- [89] I. Steinwart, “Sparseness of support vector machines—some asymptotically sharp bounds,” in *Advances in Neural Information Processing Systems 16*, S. Thrun, L. K. Saul, and B. Schölkopf, Eds. Cambridge, MA: MIT Press, 2004.
- [90] T. Hastie, R. Tibshirani, and J. Friedman, *The elements of statistical learning: data mining, inference and prediction*, 2nd ed. Springer, 2009. [Online]. Available: <http://www-stat.stanford.edu/~tibs/ElemStatLearn/>
- [91] I. G. Mayowa Aregbesola, “Augmented lagrangian fast projected gradient algorithm with working set selection for training support vector machines,” *J. Appl. Numer. Optim.*, vol. 3, no. 1, pp. 3–20, 2021. [Online]. Available: <http://jano.biemdas.com/archives/1224>

Curriculum Vitae

Mayowa Aregbesola grew up in Ile Ife, Nigeria. He received his Bachelor of Science degree in Electronic and Electrical Engineering from Obafemi Awolowo University in 2001. He obtained a Master of Science in Telecommunications from King Fahd University of Petroleum and Minerals in 2005 and a Master of Science in Electrical Engineering from George Mason University in 2009. He is a Wireless Communications Engineer and has worked in the Telecommunications field.