2016

Fast Online Training of L1 Support Vector Machines

Gabriella A. Melki

FAST ONLINE TRAINING OF L1 SUPPORT VECTOR MACHINES

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science at Virginia Commonwealth University.

by

GABRIELLA MELKI

B.Sc. Computer Science, American University of Beirut, 2011

Director: Vojislav Kecman,

Professor, Department of Computer Science

Director: Alberto Cano,

Assistant Professor, Department of Computer Science

Virginia Commonwealth University

Richmond, Virginia

April, 2016

Acknowledgements

I would first like to thank my thesis adviser, Dr. Vojislav Kecman, of the Department of Computer Science at Virginia Commonwealth University. He taught and guided me throughout this research project, allowing the work to be my own while steering me in the right direction whenever he thought I needed it. He not only provided me with insight on theory, but also work ethic and invaluable life advice.

I would also like to acknowledge Dr. Alberto Cano of the Department of Computer Science and Dr. Paul Brooks of the Department of Statistical Sciences and Operations Research at Virginia Commonwealth University as the advising committee for this thesis. I am very grateful for their time and comments on the work presented.

Finally, I would like to express my gratitude to my parents, my brother, and to my person, for continuously supporting and encouraging me throughout my studies and research. This accomplishment would not have been possible without them. Thank you.

TABLE OF CONTENTS

Chapter Page

LIST OF ALGORITHMS

LIST OF TABLES

Table Page

LIST OF FIGURES

Abstract

FAST ONLINE TRAINING OF L1 SUPPORT VECTOR MACHINES

By Gabriella Melki

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science at Virginia Commonwealth University.

Virginia Commonwealth University, 2016.

Directors: Vojislav Kecman, Alberto Cano

Department of Computer Science

This thesis proposes a novel experimental environment (non-linear stochastic gradient descent, NL-SGD), as well as a novel online learning algorithm (OL SVM), for solving a classic nonlinear Soft Margin L1 Support Vector Machine (SVM) problem using a Stochastic Gradient Descent (SGD) algorithm. The NL-SGD implementation has a unique method of random sampling and alpha calculations. The developed code produces a competitive accuracy and speed in comparison with the solutions of the Direct L2 SVM obtained by software for Minimal Norm SVM (MN-SVM) and Non-Negative Iterative Single Data Algorithm (NN-ISDA). The latter two algorithms have shown excellent performances on large datasets; which is why we chose to compare NL-SGD and OL SVM to them. All experiments have been done under strict double (nested) cross-validation, and the results are reported in terms of accuracy and CPU times. OL SVM has been implemented within MATLAB and is compared to the classic Sequential Minimal Optimization (SMO) algorithm implemented within MATLAB's solver, *fitcsvm*. The experiments with OL SVM have been done using k-fold cross-validation and the results reported in % error and % speedup of CPU Time.

CHAPTER 1

INTRODUCTION

Over the past decade, dataset sizes have grown faster and disproportionately to the speed of processors and memory capacity. With this prominent increase of large-scale data, there is a demand for new machine learning algorithms that are able to process these data to provide insightful information while completing these tasks quickly and feasibly (because of memory constraints). SVMs represent a set of popular supervised, linear and nonlinear, machine learning algorithms having theoretical foundations based on Vapnik-Chervonenkis theory [6, 16]. SVM models have similarities to other machine learning techniques, but research has shown that they usually outperform them in terms of computational efficiency, scalability, robustness against outliers, and most importantly they are well suited for ultra-large data sets [16, 9, 24, 20], making them a very useful data mining tool for diverse real-world applications. A traditional approach to training SVMs is the Sequential Minimal Optimization (SMO) technique [15], which is an efficient approach for solving Quadratic Programming tasks while training the L1 SVM. More recent and efficient approaches include the Iterative Single Data Algorithm (ISDA) [6, 9]. It has been shown that ISDA, applied as the L2 SVM solver, is faster than the SMO algorithm, and equal in terms of accuracy [23]. Here, we will show that the SGD approach is faster than both MN-SVM and NN-ISDA. MN-SVM belongs to the class of geometric approaches for solving an SVM training task and was proposed to remedy the speed of learning issues [20]. However, the newest algebraic solution of the L2 SVMs by NN-ISDA seems to be the fastest approach for dealing with large datasets [9, 24, 8]. More recently, there has been more investigation into stochastic, also known as online, algorithms to optimize large- scale learning problems. In [10] and [1] it has been shown that stochastic algorithms can be both the fastest, and have the best generalization performances. Bottou [1] and Shalev-Shwartz et al. [18, 17] demonstrate that the basic Stochastic Gradient Descent (SGD) algorithm is very effective when the data is sparse, taking less than linear $[O(d)]$ time and space per iteration to optimize a system with d parameters. It can greatly surpass the performance of more sophisticated batch methods on large data sets. It is also shown in [7], that online algorithms performances surpass that of the SMO algorithm, within the MATLAB platform. This thesis first shows the performances of the SGD algorithm as proposed in [17] while varying the parameters of the experiment. Unlike in [17], where the sample to be updated is selected randomly and uniformly within the main training algorithm, we shuffle the data and select the samples cyclically. We have also experimented with three different ways of calculating the final values of dual variables $\boldsymbol{\alpha_i}$, as well as different configurations for looping through the data during training. We also present a novel method of training L1 SVMs without performing kernel calculations at each iteration, thus speeding up training time. The thesis is organized as follows. We first define the notation used throughout the thesis, as well as define our problem. We then provide background information on the Gradient Descent and Stochastic Gradient Descent procedures. We then present the Hard-Margin and Soft-Margin L1 SVM problem as well as the L1 SVM dual formulation. This is then followed by the Soft-SVM SGD algorithm in [17], with and without using kernels. Our algorithms are then presented, followed by an explanation of our experimental environment, and a presentation of our results and how they compare with different models. Our experiments were done with nested cross-validation, ensuring reliable model comparisons. Finally, we present the conclusions and discuss our future works.

CHAPTER 2

BACKGROUND

2.1 Notation and Problem Definition

We first will formally describe the classification problem and define the notation that will be used in the paper for the description and definition of our contributions. Let D be a training dataset of N instances that is generated i.i.d from a distribution. Let C_m be the set of class labels existing describing D, where m is the number of class labels. Let X be a random vector consisting of d input variables, sometimes called input space, $X_1, \dots, X_d$, having a domain of $X \in \mathbb{R}^d$. Let Y be a label vector containing the class labels for the input vectors, sometimes called output space, having a domain of $Y^{(l)} \in C$, $l = 1, \dots, m$. For each sample $(x^{(l)}, y^{(l)})$, $x^{(l)} = \{x_1^{(l)}, \dots, x_i^{(l)}, \dots, x_d^{(l)}\}$ is the input vector and $y^{(l)} \in C$ is the class label for $x^{(l)}$, where $l \in \{1, \dots, N\}$, and $i \in \{1, \dots, d\}$. Using the training dataset $D = \{(x^{(1)}, y^{(1)}), \dots, (x^{(N)}, y^{(N)})\}$, the goal is to learn a classification model $h : X \times Y$, that assigns a label y with m possible class values, for each input instance $\mathbf{x}$. The model will then be used to simultaneously predict the values of $\{y^{N+1}, \dots, y^{N'}\}$ for new, unlabeled input vectors $\{\mathbf{x}^{N+1}, \dots, \mathbf{x}^{N'}\}$.

2.2 Regularized Loss Minimization

Regularized Loss Minimization (RLM) is a learning rule where a loss and regularization function, also known as a regularizer, are minimized together. This is given by

$$\underset{w}{\operatorname{argmin}} \quad \mathcal{R} = \left(\frac{\lambda}{2} ||w||^2 + L_S(w) \right), \tag{2.1}$$

where $L_S(\boldsymbol{w})$ is the loss function, and $\frac{\lambda}{2}||w||^2$ is the regularizer.

There are many different possible regularizers, but for the scope of this thesis we will use $R(\boldsymbol{w}) = \frac{\lambda}{2}||\boldsymbol{w}||^2$, where $\lambda > 0$ is a scalar, and the norm of $\boldsymbol{w}$ is the l_2 norm, $||\boldsymbol{w}|| = \sqrt{\sum_{i=1}^{d} w_i^2}$. This is called Tikhonov regularization, and it acts as a stabilizer for the learning rule. Stable learning rules do not overfit models to the data, where the tradeoff between fitting and overfitting is dependent upon the parameter λ. The main property of this type of regularization is that, for a convex $L_S(\boldsymbol{w})$, the objective function of RLM becomes λ-strongly convex, as shown in [17].

In the next section, we will describe the Stochastic Gradient Descent procedure and how it can efficiently solve convex problems such as the RLM rule.

2.3 Stochastic Gradient Descent

In the context of convex learning problems, such as RLM, the advantage of SGD is that it is an efficient and simple algorithm that has a good sample complexity. In this section we will first describe gradient descent, subgradients for non-differentiable function, and finally describe the SGD algorithm with some variants.

2.3.1 Gradient Descent

The approach for minimizing a convex, differentiable function $f(\boldsymbol{w})$ is an iterative one where the gradient of f is taken at each step and used to update the variable to be minimized. The gradient of function $f : \mathbb{R}^d \to \mathbb{R}$ at $\boldsymbol{w}$, is denoted and defined by $\bigtriangledown f(\boldsymbol{w}) = \left(\frac{\partial f(\boldsymbol{w})}{\partial w[1]}, \ldots, \frac{\partial f(\boldsymbol{w})}{\partial w[d]}\right)$, the vector of partial derivatives of f with respect to $\boldsymbol{w}$. The Gradient Descent (GD) algorithm is as follows: $\boldsymbol{w}$ is first initialized to some random value (say $\boldsymbol{w}^{(1)} = 0$), and at each iteration, a step is taken in the negative

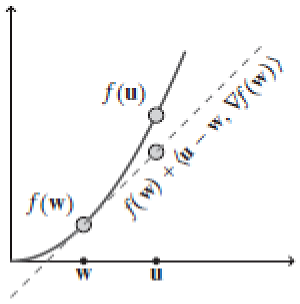
$f(\mathbf{u})$
$f(\mathbf{w})$
$f(\mathbf{w})+\langle \mathbf{u}-\mathbf{w}, \nabla f(\mathbf{w})\rangle$
w
u

Fig. 1. Right-hand side of Equation 2.3 is the tangent of f at $\boldsymbol{w}$, where for convex functions, the tangent lower bounds f [17].

direction of the gradient at that point, as shown in equation 2.2.

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \eta \bigtriangledown f(\boldsymbol{w}^{(t)}) \tag{2.2}$$

After T iterations, the algorithm returns the averaged weight vector $\bar{\boldsymbol{w}} = \frac{1}{T}\sum_{t=1}^{T} \boldsymbol{w}^{(t)}$. Different versions of the weight could also be returned, such as the last weight vector or the average of the last 25% of changes, but this will be elaborated on in section 2.4.4. The scalar parameter $\eta > 0$ will also be discussed later.

2.3.2 Subgradients

The Gradient Descent algorithm requires that the function being minimized is differentiable. Some loss functions are not differentiable, and in these cases, the subgradient of $f(\boldsymbol{w})$ at $\boldsymbol{w}^{(t)}$ can be taken instead of the gradient. A vector $\boldsymbol{v}$ that satisfies Equation 2.3 is called a *subgradient* of f at $\boldsymbol{w}$ and the set of subgradients is called the *differential set* denoted as $\delta f(\boldsymbol{w})$ [17]. For every $\boldsymbol{w} \in S$, where S is an open convex set, there exists a $\boldsymbol{v}$ such that

$$f(\boldsymbol{u}) \geq f(\boldsymbol{w}) + \langle \boldsymbol{u} - \boldsymbol{w}, \boldsymbol{v} \rangle, \, \forall \boldsymbol{u} \in S. \tag{2.3}$$

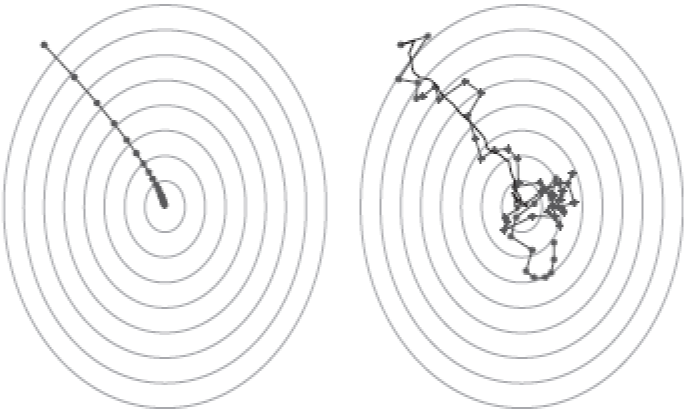

Fig. 2. A plot of the gradient descent algorithm (left) and the stochastic gradient descent algorithm (right) [17]. The function being minimized is $1.25(x+6)^2+(y-8)^2$. In the figure representing stochastic gradient descent, the solid line shows the averaged value of $\boldsymbol{w}$.

2.3.3 Stochastic Gradient Descent

For the stochastic gradient descent algorithm, the update direction does not need to be based exactly on the gradient. It is instead allowed to be a random vector, and at each iteration, its *expected value* is required to be a subgradient of the function a the current vector. An example of the difference between the SGD and the GD procedure for minimization of convex function is shown in Fig. 2. As stated previously in 2.2 and in [17], if a function f is λ-strongly convex for every vector $\boldsymbol{w}, \boldsymbol{u}$, and $\boldsymbol{v} \in \partial f(\boldsymbol{w})$ then we have $\langle w-u, v\rangle \geq f(w)-f(u)+\frac{\lambda}{2}||w-u||^2$ implying that the norms will never be greater than λ. It is then shown in [17] that an unbiased estimate of the function's subgradient can be chosen. The update rule for these functions can then be rewritten as 2.4.

$$w^{(t+1)} = -\frac{1}{\lambda t}\sum_{i=1}^{t} v_i, \tag{2.4}$$

where t is the number of iterations. This shows, by using SGD, the regularized loss function can directly be minimized. It is done by sampling a point i.i.d from a distribution, then using a subgradient of the loss function at this point as an unbiased estimate of the subgradient of the risk function.

2.4 Support Vector Machines

In this section, we will present a very useful learning tool, the Support Vector Machine (SVM). The SVM is particularly useful for learning linear predictors in high dimensional feature spaces, which is a computationally complex learning problem. This problem is approached (in the context of classification) by searching for the optimal maximal margin of separability between classes. It will first describe the Hard-Margin SVM in the case of linearly separable data, which will then be extended to the derivation of the Soft-Margin SVM for non-linearly separable datasets.

2.4.1 Hard-Margin SVM

Let S be a training set such that $S = \{(\boldsymbol{x_1}, y_1), \ldots, (\boldsymbol{x_i}, y_i), (\boldsymbol{x_m}, y_m)\}$, $\forall i \in \{1, \ldots, m\}$, $\boldsymbol{x_i} \in \mathbb{R}^d$, and $y_i \in \{+1, -1\}$, where $+1$ and -1 are the class labels. The training set S is *linearly separable* if a halfspace exists, $(\boldsymbol{w}, b)$, such that $y_i = d(\boldsymbol{x}) = sign(\langle \boldsymbol{w}, \boldsymbol{x_i} \rangle + b), \forall i \in \{1, \ldots, m\}$, where $\mathbf{w}$ is a d-dimensional weight vector, and b is a bias term. The sign function can be rewritten as constraints 2.5,

$$y_i \left(\langle \boldsymbol{w}, \boldsymbol{x}_i \rangle + b \right) > 0, \ \forall i \in \{1, \ldots, m\}. \tag{2.5}$$

All halfspaces defined as $d(\boldsymbol{x}_i) = \langle \boldsymbol{w}, \boldsymbol{x}_i \rangle + b$ which satisfy 2.5 have 0 error. There could be many different hypothesis that satisfy these constraints. A 2-dimensional example of different possible separating hyperplanes, that correctly classify all the data points, is shown in Figure 3. In Figure 3, two hyperplane decision boundaries are shown to correctly classify the two classes of data points. Intuitively, the solid

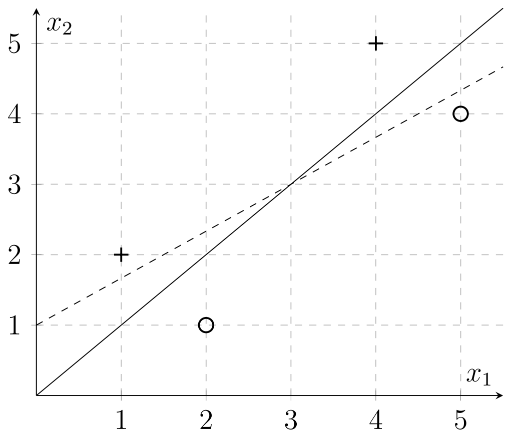

Fig. 3. Possible separating margins

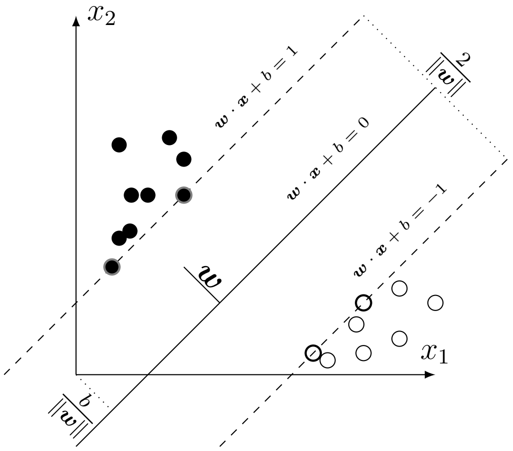
x_2
$\boldsymbol{w} \cdot \boldsymbol{x} + b = 1$
$\frac{2}{||\boldsymbol{w}||}$
$\boldsymbol{w} \cdot \boldsymbol{x} + b = 0$
$\boldsymbol{w} \cdot \boldsymbol{x} + b = -1$
$\boldsymbol{w}$
x_1
$\frac{b}{||\boldsymbol{w}||}$

Fig. 4. Maximum margin hyperplane

line seems like the better and more generalized solution, and this line of thinking can be expressed with the concept of *margin*.

The minimal distance between points belonging to opposite classes in the training set and the hyperplane is defined as the margin and has a width equal to $\frac{2}{||\boldsymbol{w}||}$. In the example shown in Figure 3, if we slightly move the training data points, the solid line (with the larger margin) will still correctly classify all the instances, whereas the dotted line (with a much smaller margin, comparatively) will not. From this, we can see that the location of the hyperplane has a direct impact on the classifiers generalization capabilities. The hyperplane with the largest margin is called the *optimal separating hyperplane*. Figure 4 shows the optimal separating hyperplane for the training data points, where the filled data points are from the $+1$ class and the non-filled data points are from the -1 class. The training data points on the separating hyperplane (the circled data points), whose decision function value equals $+1$ or -1, are called the *support vectors*.

To obtain the optimal separating hyperplane, the norm of the $\boldsymbol{w}$ vector must be minimized. This is because the margin is inversely proportional to the $\boldsymbol{w}$ vector. Therefore, to find the optimal separating hyperplane for the Hard-Margin SVM problem, one must minimize,

$$\begin{aligned} \underset{w,b}{\text{minimize}} \quad & \frac{1}{2}||\boldsymbol{w}||^2 && (2.6) \\ \text{subject to} \quad & y_i(\langle \boldsymbol{w}, \boldsymbol{x}_i \rangle + b) \geq 1, \qquad \forall i \in \{1, \dots, m\} && (2.7) \end{aligned}$$

Note, the minimization of $||\boldsymbol{w}||$ produces the same optimal $\boldsymbol{w}$ as the minimization of $||\boldsymbol{w}||^2$.

It is sometimes more convenient to consider homogeneous hyperplanes, in other words, hyperplanes that pass through the origin, where the bias term b is set to 0. We can reduce the problem of learning from non-homogeneous hyperplanes to a problem of learning from a homogeneous hyperplane by adding an extra feature to each instance of $\boldsymbol{x}_i$, increasing the input space dimensionality to $d+1$. As noted in [17], in Equation 2.6, we do not regularize the bias term b, but if we learn a homogeneous hyperplane in $\mathbb{R}^{d+1}$ as shown in Equation 2.8, we do regularize the bias term since it is the $d + 1^{th}$ component of the input vector. [17] also notes that regularizing b does not make a significant difference to the complexity of the optimization problem.

$$\min_{w} \quad ||\boldsymbol{w}||^2 \quad \text{s.t.} \quad \forall i, \quad y_i \langle \boldsymbol{w}, \boldsymbol{x}_i \rangle \geq 1 \tag{2.8}$$

Since our dataset S is assumed to be linearly separable, there exists a $\boldsymbol{w}$ and b that satisfy constraints 2.7. There will be no feasible solution for the Hard-Margin SVM problem if the dataset is non-linearly separable, which is the case in most practical datasets. To overcome this problem, Cortes and Vapnik [4] introduced the Soft-Margin SVM problem.

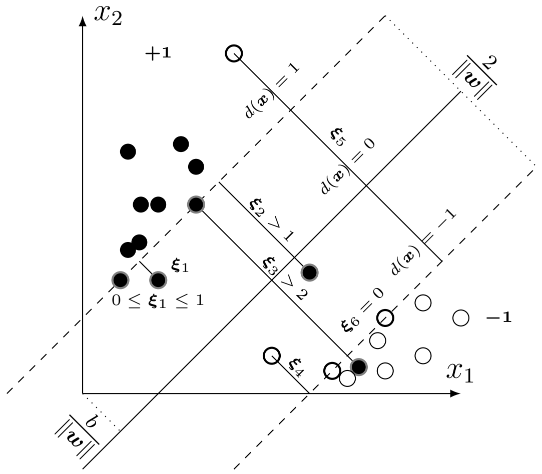

Fig. 5. 2-dimensional non-linearly separable example

2.4.2 Soft-Margin SVM

The Soft-SVM problem can be viewed as a relaxation of the Hard-SVM problem, in the sense that it can be applied to non-linearly separable data. 2.7 enforces hard constraints for all $i \in \{1, \ldots, m\}$, but in the Soft-SVM problem, it has been relaxed to allow the constraints to be violated for some of the samples in the training set by introducing non-negative slack variables ξ_i. The constraints can now be written as,

$$y_i \left(\langle \boldsymbol{w}, \boldsymbol{x}_i \rangle + b \right) \geq 1 - \xi_i, \quad \forall i \in \{1, \ldots, m\}. \tag{2.9}$$

Figure 5 shows an example of training data that is not linearly separable in input space. For the data points that lie in the margin, such as data points with corresponding slack variables ξ_1 and ξ_4, have slack variable values between 0 and 1. They are correctly classified, but do not have the maximum margin. Data points having slack variable values greater than 1 are misclassified because they lie on the wrong side of the decision boundary. The slack variables allow some error in the misclassification to account for overlapping data sets.

Due to the addition of the slack variables, the Soft-Margin SVM problem can now be re-written as,

$$\begin{aligned}
\underset{w,\xi}{\text{minimize}} \quad & \frac{\lambda}{2}||\boldsymbol{w}||^2 + \frac{1}{m}\sum_{i=1}^{m}\xi_i^p, & & \qquad (2.10)\\
\text{subject to} \quad & y_i(\langle \boldsymbol{w}, \boldsymbol{x}_i\rangle + b) \geq 1 - \xi_i, & \forall i \in \{1, \ldots, m\} & \qquad (2.11)\\
& \xi_i \geq 0, & \forall i \in \{1, \ldots, m\} & \qquad (2.12)
\end{aligned}$$

where $\lambda > 0$ is the penalty parameter that controls the trade-off between margin maximization and classification error minimization. The λ parameter penalizes large norms as well as errors. For the context of this contribution, we will set p to be equal to 1 which provides the L1 SVM formulation. The resulting hyperplane is called *soft-margin hyperplane*.

We can rewrite Equations 2.10 and 2.11 as a regularized loss minimization problem using the hinge loss function given in Equation 2.13.

$$L_S^{hinge}(\boldsymbol{w}, b) = \frac{1}{m}\sum_{i=1}^{m}\max\{0, 1 - y_i\langle \boldsymbol{w}, \boldsymbol{x}_i\rangle + b\} \qquad (2.13)$$

Then, given $(\boldsymbol{w}, b)$ and a training set S, we can consider the following optimization problem,

$$\underset{w,b}{\text{minimize}} \quad \left(\frac{\lambda}{2}||\boldsymbol{w}||^2 + L_S^{hinge}(\boldsymbol{w}, b)\right) \qquad (2.14)$$

We can say that equations Equation 2.14 and Equations 2.10 to 2.12 are equivalent as shown in [17]. We can now see that the Soft-SVM problem falls into the regularized loss minimization paradigm presented in Section 2.2. The hinge loss function is a convex one; it is not differentiable but has a subgradient, which means that it can be minimized computationally efficiently.

It is more convenient to consider the Soft-SVM problem for learning a homogeneous hyperplane, as stated above, which produces the following optimization problem,

$$\underset{w}{\text{minimize}} \quad \left(\frac{\lambda}{2}||\boldsymbol{w}||^2 + L_S^{hinge}(\boldsymbol{w})\right) \tag{2.15}$$

where

$$L_S^{hinge}(\boldsymbol{w}) = \frac{1}{m}\sum_{i=1}^{m}\max\{0, 1 - y_i\langle \boldsymbol{w}, \boldsymbol{x}_i\rangle\}. \tag{2.16}$$

2.4.3 L1-SVM and Duality

Many properties of SVM can be obtained by considering the dual of Equation 2.15. The context of this thesis does not rely on the dual formulation of 2.15, but for completeness we will present the dual formulation.

The L1 SVM is a type of SVM that has the value of p equal to 1.

$$\underset{w,\xi}{\text{minimize}} \quad \frac{1}{2}||\boldsymbol{w}||^2 + \frac{C}{m}\sum_{i=1}^{m}\xi_i, \tag{2.17}$$

$$\text{subject to} \quad y_i(\langle \boldsymbol{w}, \boldsymbol{x}_i\rangle + b) \geq 1 - \xi_i, \qquad \forall i \in \{1, \ldots, m\} \tag{2.18}$$

$$\xi_i \geq 0, \qquad \forall i \in \{1, \ldots, m\} \tag{2.19}$$

Given the above optimization problem, where $C = \frac{1}{\lambda}$ is the SVM penalty parameter, the dual can be found by first forming the primal Lagrangian given in Equation 2.20

$$\mathcal{L}_p\left(\boldsymbol{w}, b, \boldsymbol{\xi}, \boldsymbol{\alpha}, \boldsymbol{\beta}\right) = \tfrac{1}{2}||\boldsymbol{w}||^2 + \tfrac{C}{m}\textstyle\sum_{i=1}^{m}\xi_i - \sum_{i=1}^{m}\boldsymbol{\alpha}_i\left(y_i(\langle \boldsymbol{w}, \boldsymbol{x}_i\rangle + b) - 1 + \xi_i\right) - \sum_{i=1}^{m}\boldsymbol{\beta}_i\xi_i, \tag{2.20}$$

where $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ are the non-negative Lagrange multipliers.

$$\boldsymbol{\alpha_i} \geq 0, \qquad \forall i \in \{1, \ldots, m\} \tag{2.21}$$

$$\boldsymbol{\beta_i} \geq 0, \qquad \forall i \in \{1, \ldots, m\} \tag{2.22}$$

The following Karush-Kuhn-Tucker (KKT) conditions must be satisfied:

$$\frac{\partial \mathcal{L}}{\partial w_j} = 0 \qquad , \forall j \in \{1, \ldots, d\} \tag{2.23}$$

$$\frac{\partial \mathcal{L}}{\partial b} = 0 \tag{2.24}$$

$$\frac{\partial \mathcal{L}}{\partial \xi_j} = 0 \qquad , \forall j \in \{1, \ldots, d\} \tag{2.25}$$

$$\boldsymbol{\alpha}_i \left(y_i(\langle \boldsymbol{w}, \boldsymbol{x}_i \rangle + b) - 1 + \xi_i \right) = 0, \qquad \forall i \in \{1, \ldots, m\} \tag{2.26}$$

$$\boldsymbol{\beta}_i \xi_i = 0, \qquad \forall i \in \{1, \ldots, m\} \tag{2.27}$$

$$\boldsymbol{\alpha}_i \geq 0,\ \boldsymbol{\beta}_i \geq 0,\ \xi_i \geq 0, \qquad \forall i \in \{1, \ldots, m\} \tag{2.28}$$

At optimality, $\bigtriangledown_{w,b,\xi} \mathcal{L}(\boldsymbol{w}, \boldsymbol{b}, \boldsymbol{\xi}) = 0$, and the following conditions are met:

$$\frac{\partial \mathcal{L}}{\partial w_j} : \boldsymbol{w}_j = \sum_{i=1}^{m} \boldsymbol{\alpha}_i \, y_i \, x_{i\,j}, \qquad \forall j \in \{1, \ldots, d\} \tag{2.29}$$

$$\frac{\partial \mathcal{L}}{\partial b} : \sum_{i=1}^{m} \boldsymbol{\alpha}_i y_i = 0, \qquad \forall i \in \{1, \ldots, m\} \tag{2.30}$$

$$\frac{\partial \mathcal{L}}{\partial \xi_j} : \boldsymbol{\alpha}_i + \boldsymbol{\beta}_i = \frac{C}{m}, \qquad \forall j \in \{1, \ldots, d\},\ \forall i \in \{1, \ldots, m\} \tag{2.31}$$

By substituting Equations 2.29, 2.30, and 2.31 into the Lagrangian in 2.20, the following dual problem is obtained:

$$\underset{\alpha}{\text{minimize}} \quad \frac{1}{2} \sum_{i,j=1}^{m} \boldsymbol{\alpha}_i \boldsymbol{\alpha}_j y_i y_j \langle \boldsymbol{x_i}, \boldsymbol{x_j} \rangle - \sum_{i=1}^{m} \boldsymbol{\alpha}_i \tag{2.32}$$

$$\text{subject to} \quad \sum_{i=1}^{m} \boldsymbol{\alpha}_i y_i = 0, \tag{2.33}$$

$$0 \leq \boldsymbol{\alpha}_i \leq \frac{C}{m}, \qquad \forall i \in \{1, \ldots, m\} \tag{2.34}$$

There are three cases that are possible for the α_i values, from the constraints given in Equations 2.26, 2.27, and 2.28, given below:

1. If $\boldsymbol{\alpha}_i = 0$, then $\boldsymbol{\beta}_i = \frac{C}{m}$ and that $\xi_i = 0$, which indicates a correctly classified

instance outside the margin.

2. If $0 \leq \boldsymbol{\alpha}_i \leq \frac{C}{m}$, then $\boldsymbol{\beta}_i > 0$ and $\xi_i = 0$, indicating that a point sits on the margin boundary. This instance will be an *unbounded support vector*.

3. If $\boldsymbol{\alpha}_i = \frac{C}{m}$, then $\boldsymbol{\beta}_i = 0$ and there is no restriction for $\xi_i \geq 0$. This also indicates that the instance is a support vector that is *unbounded*. If $0 \leq \xi_i < 1$, then the instance is correctly classified, otherwise it is misclassified.

2.4.4 Stochastic Gradient Descent for the Soft-SVM Problem

The Soft Margin SVM optimization problem associated with regularized hinge loss minimization, minimizes the following cost function in Equation 2.15, which is λ-strongly convex. In this section, we will describe the SGD procedure in [17] for solving the Soft-SVM problem.

Recall that we can rewrite the update rule in SGD as Equation 2.4 given below,

$$\boldsymbol{w}^{(t+1)} = -\frac{1}{\lambda\, t} \sum_{i=1}^{t} \boldsymbol{v}_i,$$

where, at iteration $j < t$, $\boldsymbol{v}_i$ is a subgradient of the loss function at $\boldsymbol{w}^{(j)}$ on the random sample chosen. For the hinge loss function, we choose v to be

$$\boldsymbol{v}_j := \begin{cases} 0, & \text{if } y\langle \boldsymbol{w}^{(j)}, \boldsymbol{x}\rangle \geq 1 \\ -y\boldsymbol{x}, & \text{otherwise.} \end{cases} \tag{2.35}$$

Setting $\boldsymbol{\theta}^{(t)} = -\sum_{j<t} \boldsymbol{v}_j$, Algorithm 1 minimizes Equation 2.15.

The SVM learns hyperplanes with a preference for a large margin. The Hard-SVM finds the hyperplane that separates data perfectly with the largest possible margin, whereas the Soft-SVM does not assume separability and allows the constraints to be violated with some error. The sample complexity of finding this hyperplane is

Algorithm 1 SGD for Solving Soft-SVM

```
Input: Training dataset $D$, number of cross-validation folds $k$, parameter $T$
Output: Optimal SVM weight vector $\mathbf{w}^*$
Initialize $\boldsymbol{\theta}^{(1)} = \mathbf{0}$
for $t = 1$ to $T$ do
    Let $\mathbf{w}^{(t)} = \frac{1}{\lambda t}\boldsymbol{\theta}^{(t)}$
    Choose $i$ uniformly at random from $[m]$
    if $y_i\langle \mathbf{w}^{(t)}, \mathbf{x}_i\rangle < 1$ then
        Set $\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} + y_i\mathbf{x}_i$
    else
        Set $\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)}$
    end if
end for
return $\boldsymbol{w}^* = \frac{1}{T}\sum_{t=1}^{T}\mathbf{w}^{(t)}$
```

not dependent on the dimensionality of the input space, but rather on the maximal norm of $\boldsymbol{w}$. Dimension-independent complexity is important, and in section 2.4.5 we will discuss mapping our input space into a higher-dimensional feature space in order to expand our hypothesis class to include non-linear classifiers.

2.4.5 Kernel Methods

Although the Soft-SVM learns an optimal hyperplane, if the training data set is not linearly separable, the classifier learned may not have a good generalization capability. Generalization and linear separability can be enhanced by mapping the original input space to a higher dimensional dot-product feature space by using a kernel function shown in Equation

$$K\left(\boldsymbol{x}_i, \boldsymbol{x}_j\right) = \phi\left(\boldsymbol{x}_i\right) \cdot \phi\left(\boldsymbol{x}_j\right), \tag{2.36}$$

where $\phi(\cdot)$ represents a mapping function into a higher dimensional feature space.

To solve the optimization problem given in Equation 2.32, we do not need to know the values of the elements in feature space, all we need is a method to calculate their inner products in the feature space, namely the kernel function. This means that we only need the value of the $(m \times m)$ matrix G such that $G_{ij} = K(\boldsymbol{x}_i, \boldsymbol{x}_j)$; and thus, Equation 2.32 can be rewritten (in matrix form) as:

$$\underset{\alpha}{\text{minimize}} \left(\lambda \boldsymbol{\alpha}^T G \boldsymbol{\alpha} + \frac{1}{m} \sum_{i=1}^{m} max\{0, 1 - y_i(G\boldsymbol{\alpha})_i\} \right) \tag{2.37}$$

where $(G\boldsymbol{\alpha})_i$ is the i^{th} element of the vector obtained by multiplying the matrix G by the vector $\boldsymbol{\alpha}$. Note that Equation 2.15 is a convex quadratic programming problem and can be solved efficiently. Our contribution is an even simpler method that solves this problem and is described in Chapter 3.

The advantage of working with kernels instead of optimizing $\boldsymbol{w}$ directly in feature space is that sometimes the dimensionality of the feature space is very large, and this would require extensive computation, while computing the kernel is much simpler. Table 1 shows some popular kernel functions. In the example of the Gaussian Radial Basis Function (RBF) kernel, the feature space is infinitely dimensional, while the kernel calculation is very straight-forward.

2.4.6 SGD with Kernels

While the Algorithm 1 is simple, it only works for linear SVMs. For a general nonlinear SVM (NL-SVM) model, with use of kernels, there is also a simple method for solving the Soft-SVM task by using a stochastic gradient approach. (The presentation

Table 1. Popular Kernel Functions

Kernel Function	**Name**	**Properties**
$K(\boldsymbol{x}_i, \boldsymbol{x}_j) = (\boldsymbol{x}_i \cdot \boldsymbol{x}_j)$	Linear	CPD [1]
$K(\boldsymbol{x}_i, \boldsymbol{x}_j) = (\boldsymbol{x}_i \cdot \boldsymbol{x}_j + 1)^d$	Polynomial of degree d	PD [2]
$K(\boldsymbol{x}_i, \boldsymbol{x}_j) = (e^{-\gamma \|\|\boldsymbol{x}_i - \boldsymbol{x}_j\|\|^2})$	Gaussian RBF	PD [2]

[1] Conditionally Positive Definite
[2] Positive Definite

below follows [17]). For NL-SVM, one wants to minimize the regularized soft margin loss function,

$$\min_{w} \left(\frac{\lambda}{2} ||\boldsymbol{w}||^2 + \frac{1}{m} \sum_{i=1}^{m} \max\{0, 1 - y_i \langle \boldsymbol{w}, \phi(\boldsymbol{x}_i) \rangle\} \right) \tag{2.38}$$

where vector $\boldsymbol{w}^{(t)}$ that is updated with Algorithm 1 is always in the linear span of $\{\phi(\boldsymbol{x}_1), \phi(\boldsymbol{x}_2), \ldots, \phi(\boldsymbol{x}_m)\}$, which means that the corresponding coefficient $\boldsymbol{\alpha}$ can be maintained. In this new SGD procedure, [17] maintains two vectors $\boldsymbol{\beta}^{(t)}$ and $\boldsymbol{\alpha}^{(t)}$ that are updated by Algorithm 2 such that,

$$\boldsymbol{\theta}^{(t)} = \sum_{j=1}^{m} \boldsymbol{\beta}_j^{(t)} \phi(\boldsymbol{x}_j) \tag{2.39}$$

$$\boldsymbol{w}^{(t)} = \sum_{j=1}^{m} \boldsymbol{\alpha}_j^{(t)} \phi(\boldsymbol{x}_j). \tag{2.40}$$

The output of both Algorithm 1 and Algorithm 2 are equal. Let $\hat{\boldsymbol{w}}$ be the output of Algorithm 1 in feature space, and let $\bar{\boldsymbol{w}} = \sum_{j=1}^{m} \bar{\boldsymbol{\alpha}}_j \phi(\boldsymbol{x}_j)$ be the output of Algorithm 2. For every iteration t, by definition $\boldsymbol{\alpha}^{(t)} = \frac{1}{\lambda t} \boldsymbol{\beta}^{(t)}$ and $\boldsymbol{w}^{(t)} = \frac{1}{\lambda t} \boldsymbol{\theta}^{(t)}$ which implies that Equation 2.40 holds. For proving that Equation 2.39 holds, our

Algorithm 2 SGD for Solving Soft-SVM with Kernels

```
**Input:** Training dataset $D$, number of cross-validation folds $k$, parameter $T$
**Output:** Optimal SVM weight vector $\mathbf{w}^*$
Initialize $\boldsymbol{\beta}^{(1)} = 0$
for $t = 1$ to $T$ do
    Let $\boldsymbol{\alpha}^{(t)} = \frac{1}{\lambda m}\boldsymbol{\beta}^{(t)}$
    Choose $i$ uniformly at random from $[m]$
    $\forall j \neq i$, set $\boldsymbol{\beta}_j^{(t+1)} = \boldsymbol{\beta}_j^{(t)}$
    if $y_i \sum_{j=1}^{m} \boldsymbol{\alpha}_j^{(t)} \boldsymbol{K(x_j, x_i)} < 1$ then
        Set $\boldsymbol{\beta}^{(t+1)} = \boldsymbol{\beta}^{(t)} + y_i$
    else
        Set $\boldsymbol{\beta}^{(t+1)} = \boldsymbol{\beta}^{(t)}$
    end if
end for
return $\mathbf{w}^* = \sum_{j=1}^{m} \boldsymbol{\alpha}_j^* \, \phi(x_j)$, where $\boldsymbol{\alpha}^* = \frac{1}{T}\sum_{t=1}^{T} \boldsymbol{\alpha}^{(t)}$
```

base case where $t = 1$ trivially holds. For $t \geq 1$,

$$y_i \langle \boldsymbol{w}^{(t)}, \phi(\boldsymbol{x}_i) \rangle = y_i \langle \sum_j \boldsymbol{\alpha}_j^{(t)} \phi(\boldsymbol{x}_j) \phi(\boldsymbol{x}_i) \rangle = \sum_{j=1}^{m} \boldsymbol{\alpha}_j^{(t)} \boldsymbol{K}(\boldsymbol{x}_j, \boldsymbol{x}_i). \tag{2.41}$$

This shows that the two algorithms are equivalent, and when updating $\boldsymbol{\theta}$,

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} + y_i \phi(\boldsymbol{x}_i) = \sum_{j=1}^{m} \boldsymbol{\beta}^{(t)} \phi(\boldsymbol{x}_j) + y_i \phi(\boldsymbol{x}_i) = \sum_{j=1}^{m} \boldsymbol{\beta}^{(t+1)} \phi(\boldsymbol{x}_j). \tag{2.42}$$

To summarize, we have shown how kernels can be used in the SGD algorithm for solving the Soft-SVM optimization problem. Feature mapping and the kernel trick allows us to use linear predictors for non-linear data. In the next chapter, we will present a simple method and experimental environment for solving the L1 SVM problem that has competitive accuracy and very fast CPU time for datasets of any size. We will then compare our method to two competitive methods for solving the SVM problem, a geometric and an iterative approach. We will also present a different formulation for performing updates in the SGD procedure that also speeds up training time by using a slightly different loss function during optimization, as well as compare it's performance to the popular SMO algorithm.

CHAPTER 3

STOCHASTIC GRADIENT DESCENT ALGORITHMS FOR SOLVING SVMS

In this Chapter, we will present a modified version of the SGD algorithm for solving the L1 SVM problem presented as Algorithm 2. The presented algorithm, *Non-linear SGD* (NL-SGD), was developed and tested within the GSVM framework [21]. We will then present a variant of the RLM problem for solving SVMs in the primal domain along with a novel approach [7] called *On-Line Support Vector Machine* (OL SVM) for optimizing it.

3.1 Non-Linear SGD

The first modification made to Algorithm 2 was changing the method of sampling. To improve the computational complexity of the algorithm, the data is first shuffled and then learning can then proceed cyclically. Rather than produce a random index at ever iteration, no random number generation is needed in the NL-SGD method because the data is already shuffled. By excluding the random sampling from the algorithm's main loop, learning can be performed faster. Another advantage (besides computational complexity) of shuffling the data rather than random sampling is that shuffling ensures that each data point gets used in the learning process, therefore no knowledge can be lost. With random number generation at each iteration, there is no guarantee that each of the data points gets used in training, while with our shuffling method it does.

In Algorithm 2, the calculation of the final dual variables is done by taking the average $\boldsymbol{\alpha}$ value over all iterations. Our method, presented in Algorithm 3, tests three different approaches of calculating the final $\boldsymbol{\alpha}$ value. The first is averaging over the last 50% of the updates, the second is taking the mean value of the dual variables over the last 25% of updates, and lastly we used only the last value of $\boldsymbol{\alpha}$. The reasoning behind these different methods is that as the iterations proceed, the optimal objective function value is being approached. Taking the average of all the different changes of the $\boldsymbol{\alpha}$ values might include some added noise in the final model from previous values of $\boldsymbol{\alpha}$ that produce objective function values further from the optimal one. Averaging over later sets of the $\boldsymbol{\alpha}$ changes would ensure that no noise would be taken into account in the final model. These more complex averaging techniques can improve the convergence speed in some situations, such as in the case of strongly convex functions.

Finally, we conducted some preliminary experiments for clarifying issues associated with stopping the training process. We first decided that the number of iterations, T, would be set to the number of training samples, and then we added an extra layer by specifying how many times, or epochs, to loop over the dataset. When an epoch greater than one is used, we set T to be the number of samples multiplied by the number of epochs. We experimented with using 2 and 5 epochs for small datasets, and 1 and 2 epochs for medium datasets. We used a higher number of epochs for small datasets because the number of samples might not be high enough for the minimum to be found.

Preprocessing: Before beginning initialization of the training process, we first perform some preprocessing on the datasets provided as input. We normalize the data by scaling the features values to be between $[0, 1]$. Scaling is performed to ensure that no feature has any bias (by having relatively larger values compared to other features) during the training process. We then shuffle the data in order to proceed cyclically,

Algorithm 3 Non-Linear Stochastic Gradient Descent for SVM

```
Input: Training dataset $(\boldsymbol{x_i}, y_i) \in D, \forall i \in \{1, \ldots, m\}$, number of epochs $e$, % of
    alpha changes $kt$
Output: Optimal SVM weight vector $\mathbf{w}^*$
Initialize $\boldsymbol{\beta}^{(1)} = 0$, $\boldsymbol{\alpha}^{(1)} = 0$, $\boldsymbol{T}^{(1)} = m * e$
for $t = 1$ to $T$ do
    $i = t$
    if $y_i \sum_{j=1}^{m} \boldsymbol{\alpha}_j^{(t)} \boldsymbol{K}(\boldsymbol{x_j}, \boldsymbol{x_i}) \leq 1$ then
        Set $\boldsymbol{\beta}^{(t+1)} = \boldsymbol{\beta}^{(t)} + y_i$
    else
        Set $\boldsymbol{\beta}^{(t+1)} = \boldsymbol{\beta}^{(t)}$
    end if
    $\eta = \frac{1}{\lambda t}$
    $\boldsymbol{\alpha}^{(t+1)} = \eta \, \boldsymbol{\beta}^{(t+1)}$
    if $t \geq kt * T$ then
        $i = 1$
    end if
end for
return $\mathbf{w}^* = \sum_{j=1}^{m} \boldsymbol{\alpha_j^*} \, \phi(x_j)$, where $\boldsymbol{\alpha}^* = \frac{1}{(1-kt)\,T} \sum_{t=1}^{kt*T} \boldsymbol{\alpha}^{(t)}$
```

by eliminating random sampling, during the training process.

Initialization: We first select a feasible starting point to begin the algorithm by initializing $\boldsymbol{\beta}^{(1)} = 0$ and $\boldsymbol{\alpha}^{(1)} = 0$. The number of iterations is also initialized to be the number of data points in the training set, m, multiplied by the number of epochs e. The indexing of the samples proceeds cyclically by setting our index, i, at each iteration to equal t. If the total number of iterations T exceeds the sample size, i.e.

if there is more than one epoch, we reset the index to equal 1 in order to loop back over the training set.

Identifying Violators & The Update Procedure: The algorithm proceeds to find a violator by checking if the constraints shown in equation 2.41 is violated. We have used a Gaussian RBF kernel, shown in Table 1, as our kernel function, $\boldsymbol{K}(\cdot)$. In our implementation, to improve computational complexity for vector calculations, we maintain the vectors by stacking the violators found at the head of each array, i.e. for vectors $\boldsymbol{\alpha}^{(t)}$ and $\boldsymbol{\beta}^{(t)}$, the violators are stored in indices $t \in \{1, \dots, nv\}$, where nv is the number of violators. If a violator is found, we swap the current sample index with the sample located at the $nv+1^{th}$ location of the $\boldsymbol{\alpha}^{(t)}$ and $\boldsymbol{\beta}^{(t)}$ vectors, increment the value of nv, and proceed with the vector updates over the values contained in locations $\{1, \dots, nv\}$. These updates are shown in Equations 3.2 and 3.3, where 3.2 follows Equation 2.42. This technique ensures that no unnecessary calculations will be performed during the algorithms main loop, such as multiplying η with a 0 value in the $\boldsymbol{\alpha}$ vector for non-violating samples. This also benefits kernel calculations, shown in Equation 3.1. The calculation to obtain the kernel value will only be over values in locations $\{1, \dots, nv\}$.

$$K(\boldsymbol{x}_i, \boldsymbol{x}_j) = (e^{-\gamma||\boldsymbol{x}_i - \boldsymbol{x}_j||^2}), \forall i, j \in \{1, \dots, nv\} \tag{3.1}$$

$$\begin{gathered}\sum_{j=1}^{nv} \boldsymbol{\beta}^{(t+1)} \phi(\boldsymbol{x}_j) = \sum_{j=1}^{nv} \boldsymbol{\beta}^{(t)} \phi(\boldsymbol{x}_j) + y_i \phi(\boldsymbol{x}_i) \rightarrow \\ \boldsymbol{\beta}_i^{(t+1)} = \boldsymbol{\beta}_i^{(t)} + y_i, \text{ where } i \text{ is the current sample}\end{gathered} \tag{3.2}$$

$$\boldsymbol{\alpha}_i^{(t+1)} = \eta \boldsymbol{\beta}_i^{(t+1)}, \forall i \in \{1, \dots, nv\} \tag{3.3}$$

Final $\boldsymbol{\alpha}$ Vector Calculation: If the input parameter kt is different than 1; i.e. if we do not want to take the last $\boldsymbol{\alpha}$ value as our final dual variable, during the learning phase we sum the values of $\boldsymbol{\alpha}$ at each iteration in anticipation of taking some average. We then calculate the final averaged $\boldsymbol{\alpha}$ value using Equation 3.4, where e is the number of epochs.

$$\boldsymbol{\alpha}^* = \frac{1}{(T\,(1-kt))}\sum_{i=1}^{e}\boldsymbol{\alpha}_i \tag{3.4}$$

In the next section, we will describe a different, yet similar, method for solving the SVM optimization problem. The key difference between the two algorithms is the identification of violators, where computation is sped up without loss of accuracy due to the elimination of kernel calculations at each iteration.

3.2 OL SVM

Recently, several authors have proposed the use of a standard SGD for SVMs [16, 11, 19, 17]. The approaches in [16, 11, 19, 17] are extended variants of a classic kernel perceptron algorithm shown in [5]. The main difference between the otherwise close approaches presented in [16, 11, 19, 17], and the variants presented in [7] and shown here, originate from the minimization of slightly different loss functions while training the SVM. In [16, 11, 19, 17] the cost function was the regularized hinge loss given by Equation 3.5, while in this section, the hinge loss, or soft margin loss, without regularization shown in Equation 3.6, is minimized.

$$\mathcal{R}_{rhl} = C\sum_{i=1}^{m}\max\left(0, 1 - y_i\, o(\boldsymbol{x_i})\right) + \frac{1}{2}||o||_2^2 \tag{3.5}$$

$$\mathcal{R}_{rl} = \sum_{i=1}^{m}\max\left(0, 1 - y_i\, o(\boldsymbol{x_i})\right) \tag{3.6}$$

o is the output, or decision function, of the SVM for a given input vector $\boldsymbol{x_i}$, and is the function being minimized. It is given by Equation 3.7. Note that o is calculated without the use of the bias term b.

$$o(\boldsymbol{x}) = \sum_{i=1}^{m} \boldsymbol{\alpha_i}\, \boldsymbol{K}(\boldsymbol{x}, \boldsymbol{x_i}) \tag{3.7}$$

Typically, this type of regularization is expressed by squaring the norm of the weight vector $\boldsymbol{w}$ as shown in Equation 2.14 which, geometrically, determines the width of the margin between two classes. [3] presents the justification for not using the regularization term, $\frac{1}{2}||\boldsymbol{o}||_2^2$, shown in Equation 3.5. The algorithm 4 presented in this section [7], OL SVM, was obtained by minimizing the soft margin loss function shown in Equation 3.6, and it is similar to the approach in [5]. It is also similar to the approach shown in Algorithm 3, except that it has a more efficient method of finding the dual variable $\boldsymbol{\alpha}$. The experiments performed using the algorithm in [7] confirm the statements and findings in [3] and will be shown in Chapter 4.

In addition to being very simply structured, the efficiency of the OL SVM algorithm is improved by calculating $viol$ variable with the help of the output vector $\boldsymbol{o}$ and by caching $\boldsymbol{o}$ only when there is a violation, i.e. an $\boldsymbol{\alpha_j}$ update. By using output vector $\boldsymbol{o}$, one avoids computing a dot product of m-dimensional vectors at each iteration step, making the computation of $viol$ more efficient. The bias term b is not needed when positive definite kernels are implemented, but it can be used. If b is used, whenever $viol < 1$, b is updated by Equation 3.8. Also, and only when $viol < 1$, the term $\eta\, y_i$ must be added to the right hand side of the output $\boldsymbol{o}$ in Algorithm 4. The bias updates in the SVM model output $\boldsymbol{o}$ are given in Equations 3.8 and 3.9.

$$b_i = b_i + \eta\, y_i \tag{3.8}$$

$$o(\boldsymbol{x}) = \sum_{i=1}^{m} \boldsymbol{\alpha_i}\, \boldsymbol{K}(\boldsymbol{x}, \boldsymbol{x_i}) + b \tag{3.9}$$

Algorithm 4 Online Algorithm - OL SVM for $\mathcal{R}_{HL}$

```
Input: Training dataset $(\boldsymbol{x_i}, y_i) \in D, \forall i \in \{1, \ldots, m\}$, number of epochs $e$
Output: Optimal SVM dual variable $\boldsymbol{\alpha}^*$
Initialize $\boldsymbol{o}^{(1)} = 0$, $\boldsymbol{\alpha}^{(1)} = 0$, $\boldsymbol{T}^{(1)} = m * e$, $i = 0$
for $t = 1$ to $T$ do
    $\eta = C\sqrt{\frac{2}{t}}$
    $i = i + 1$
    if $i > m$ then
        $i = 0$
    end if
    $viol = y_i \boldsymbol{o}_i^{(t)}$
    if $viol < 1$ then
        Set $\boldsymbol{o}^{(t+1)} = \boldsymbol{o}^{(t)} + \eta\, y_i\, \boldsymbol{k}(\cdot, \boldsymbol{x_i})$
        Set $\boldsymbol{\alpha}_i^{(t+1)} = \boldsymbol{\alpha}_i^{(t)} + \eta\, y_i$
    end if
end for
return $\boldsymbol{\alpha}^* = \boldsymbol{\alpha}^{(T)}$
```

CHAPTER 4

RESULTS

In this chapter we will describe the datasets used for our experimental environments to test the NL-SGD and the OL SVM algorithms. We will then describe the experimental setup and show the results obtained by both algorithms. More detailed results can be found in [7].

4.1 Data

4.1.1 Data for NL-SGD Experiments

To evaluate our experimental procedure, and the SGD algorithm, as well as compare its performance with the MN-SVM and NN-ISDA algorithms, we used benchmark datasets from both the UCI Machine Learning Repository [22] and the LIBSVM [13] sites. Table 2 lists all the datasets used to test our model. For each dataset, the table lists the number of inputs, the number of features (dimensionality), and the number of classes. Note that these datasets are the same as the datasets used in [20] and [23]. In [20] MN-SVM was compared to the BVM implementation taken from the LibCVM package [12], and in [23] the NN-ISDA algorithm was then compared to the results obtained by MN-SVM in [20].

MN-SVM was implemented in an open source framework called GSVM Command Line Tool for Geometric SVM Training [21] and showed considerable training time speedup in comparison to the L1 and L2 SVM methods from LIBSVM [2], as well as BVM from LibCVM. These comparisons are shown in [20]. It has been then shown in [23] that the NN-ISDA algorithm is faster than the MN-SVM. This is why

Table 2. Dataset Information

Dataset	**# Instances**	**# Features**	**# Classes**
Small Datasets			
Iris	150	4	3
Glass	214	9	6
Wine	178	13	3
Teach	151	5	3
Sonar	208	60	2
Dermatology	366	33	2
Heart	270	13	2
Prokaryotic	997	20	3
Eukaryotic	2427	20	4
Medium Datasets			
Optdigits	5620	64	10
Usps	9298	256	10
Reuters	11069	8315	2

we compare our implementation of the SGD algorithm with these two approaches. Note that all the three algorithms are implemented within the general framework of GSVM [21].

4.1.2 Data for OL SVM Experiments

The comparison of the two binary SVM classifiers, the SMO based *fitcsvm* and the implementation of OL SVM [7], is performed on datasets that contain up to 15,000 samples. An overview on the datasets used are presented in Table 3. The data that

Table 3. OL SVM Dataset Information

Dataset	# Instances	# Features
Small Datasets		
Cancer	198	32
Sonar	208	60
Vote	232	16
Eukaryotic	2427	20
prototask	8192	21
Reuters	11069	8315
Chess Board	8000	2
Two Normally distributed classes	15000	2

is not originally binary has been transformed into a binary classification problem by grouping the samples into two classes. For the *Eukaryotic* dataset, results are shown for classifying class 4 vs. the rest. The *Prototask* dataset is transformed into a balanced two class problem, where if $y_i \geq 89$, $\boldsymbol{x_i}$ is assigned to class 1, otherwise, the data belongs to class 2.

4.2 Experimental Environment

4.2.1 Experimental Environment for NL-SGD

The comparison of the NL-SGD, NN-ISDA, and MN-SVM models was obtained using a strict nested (a.k.a. double) cross-validation procedure. This experimental environment is computationally expensive, but it ensures that the models performances are accurately assessed. In the outer loop, the data is separated into equally sized sections; in our experiments we have chosen to use 5 sections. Each part is held out in turn as the test set, and the remaining four parts are used as the training set. In the inner loop, 5-fold cross-validation is used over the training set, where the best hyper-parameters are chosen. The best model obtained by the inner loop is then applied on the outer loops test set. This procedure ensures that the models performance is not optimistically biased as what would have occurred using single loop k-fold cross-validation.

The preprocessing steps used for our data sets were feature normalization and data shuffling. Our features were rescaled to values between $[0, 1]$, and we shuffled the data to ensure that our method is stochastic when looping through each data point.

The tuning of our model during cross-validation consists of finding the best penalty parameter C, or $\frac{1}{\lambda}$, as well as the best shape parameter γ for the Gaussian RBF kernel. The parameters were selected among 8×8 possible combinations, from the below values, which have also been used in [20, 23].

$$C \in 4^n,\ n \in \{-2, -1, \ldots, 5\} \tag{4.1}$$

$$\gamma \in 4^n,\ n \in \{-5, -4, \ldots, 2\} \tag{4.2}$$

To deal with multi-class classification problems, we used the one-vs-one, or pairwise, approach. The pairwise training procedure trains $\frac{c(c-1)}{2}$ binary classifiers, a classifier for each possible pair of classes, where c is the number of classes. During the prediction phase, a voting scheme is used where all $\frac{c(c-1)}{2}$ models predict an unseen data sample and the class that received the highest number of votes is considered to be the samples true class.

4.2.2 Experimental Environment for OL SVM

In all the simulations, found in [7], of the experiments for OL SVM, the SVM with a Gaussian kernel has been used. The results shown are obtained by performing the 5-fold cross-validation. The two design hyperparameters used where out of the following ranges, shown in Equations 4.3 and 4.4

$$C \in 10^n,\ n \in \{-4, -3, \ldots, 4\} \tag{4.3}$$

$$\sigma \in 10^n,\ n \in \{-1, 0, \ldots, 2\}, \tag{4.4}$$

where the Gaussian parameter $\sigma = \sqrt{2\,\gamma}$. These sets of 9 C and 4 σ values would lead to $36 \times 5 = 180$ runs for each dataset, which is computationally expensive for larger datasets. For example, the SMO algorithm with solver *fitcsvm* needed 2.7 days to finish the cross-validation runs for *Reuters* dataset.

4.3 Results

4.3.1 Comparison of NL-SGD with NN-ISDA and MN-SVM

Table 4 shows the accuracies achieved by our implementation of NL-SGD, by MN-SVM [20, 21], and NN-ISDA [9, 23].

Figures 6 and 7 show the accuracy comparisons of the three algorithms for the small and medium datasets. In all cases, NL-SGD performs competitively, or even surpasses the accuracy of NN-ISDA and MN-SVM.

Table 5 compares the three algorithms CPU training and shows that the NL-SGD implementation is the choice for learning from datasets. In all cases, except the *Wine* dataset, the NL-SGD implementation is fastest. From the perspective of the average CPU time, it is 1.2 times faster than NN-ISDA and 8 times faster than MN-SVM for training small datasets. For medium datasets, NL-SGD is 1.35 times faster

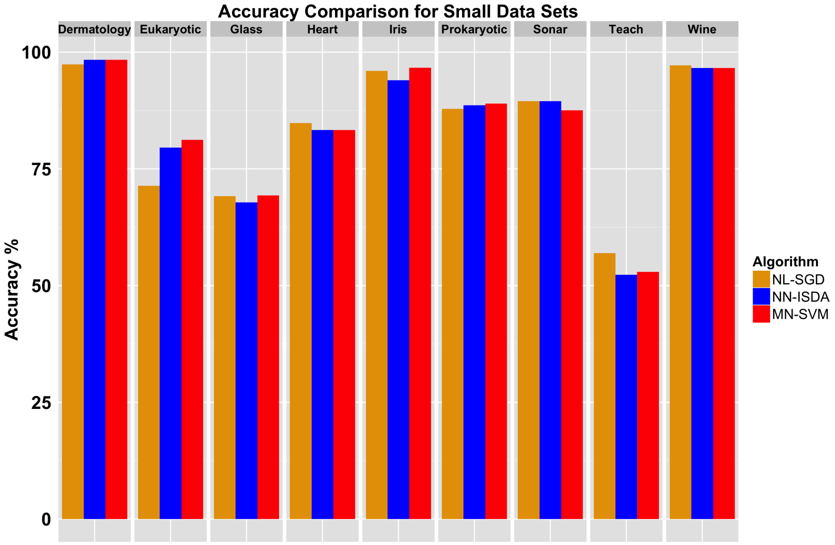

Fig. 6. Accuracy Comparison for Small Datasets

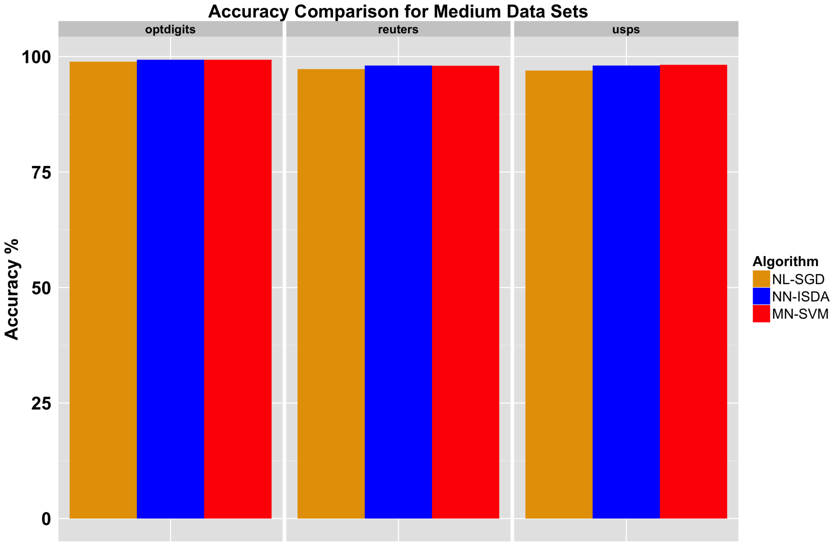

Fig. 7. Accuracy Comparison for Medium Datasets

than NN-ISDA and 1.92 times faster than MN-SVM. This, combined with NL-SGDs competitive accuracy, shows that it is definitely suitable for training datasets of any size, especially large datasets.

Table 4. NL-SGD Accuracies Achieved

Dataset	**MN-SVM**	**NN-ISDA**	**NL-SGD**
Small Datasets			
Iris	**96.67**	94.00	96.00
Glass	**69.29**	67.82	69.17
Wine	96.60	96.69	**97.17**
Teach	52.95	52.31	**56.95**
Sonar	87.57	89.48	**89.49**
Dermatology	**98.36**	**98.36**	97.38
Heart	83.33	83.33	**84.81**
Prokaryotic	**88.97**	88.65	87.86
Eukaryotic	**81.21**	79.56	71.36
Average Accuracy	**83.33**	83.34	83.35
Medium Datasets			
Optdigits	**99.31**	99.29	98.88
Usps	**98.21**	98.05	96.93
Reuters	98.05	**98.08**	97.28
Average Accuracy	**98.52**	98.47	97.69

Accuracies are reported in percents (%)

The main motivation for the experiments presented in this thesis is to examine the speed of learning that can be achieved by using the NL-SGD algorithm on datasets of increasing size. In that respect, Figs. 8, 9, 10, and Table 5 show exactly that. From the results shown in this section, we can see that our NL-SGD implementation is very competitive with respect to the NN-ISDA and MN-SVM algorithms. Being able

Table 5. NL-SGD CPU Time Needed For Training

Dataset	**MN-SVM**	**NN-ISDA**	**NL-SGD**
Small Datasets			
Iris	3.57	0.27	**0.18**
Glass	11.94	1.01	**0.5**
Wine	4.84	**0.43**	0.47
Teach	8.85	0.44	**0.47**
Sonar	3.03	0.98	**0.74**
Dermatology	11.68	2.47	**1.69**
Heart	6.45	0.91	**0.61**
Prokaryotic	50.86	10.64	**5.61**
Eukaryotic	342.76	49.16	**45.38**
Average CPU Time	49.33	7.37	**6.16**
Medium Datasets			
Optdigits	787.85	528.04	**340.70**
Usps	7777.82	5245.55	**4193.36**
Reuters	1657.04	1368.02	**758.05**
Average CPU Time	3405.57	2380.54	**1764.04**

CPU time is reported in seconds (s)

to process large amounts of data with these CPU training times provides a sizeable advantage, especially because the accuracy of the model generated, with respect to its competitors, is not reduced. The average accuracy of MN-SVM is slightly higher, but being able to build a model with significant speedup in the training time compensates this.

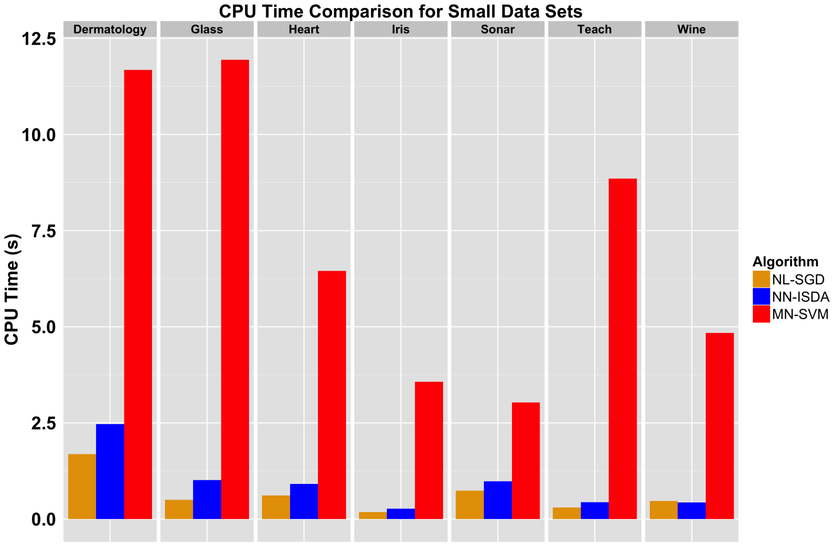

Fig. 8. CPU Time Comparison for Small Datasets

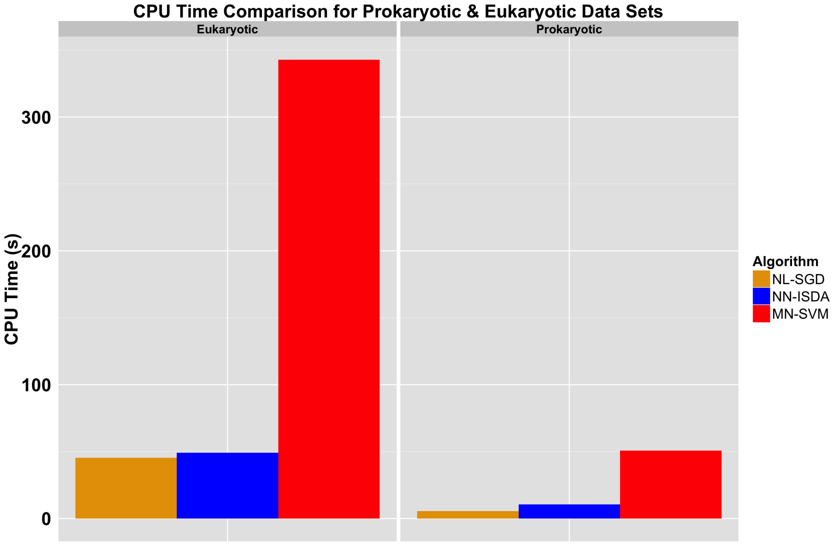

Fig. 9. CPU Time Comparison for Prokaryotic and Eukaryotic Datasets

4.3.2 Results for Different Configurations of NL-SGD

Below we can see how NL-SGDs accuracy is affected by different configurations in terms of the number of epochs chosen, as well as the percentage of alpha changes

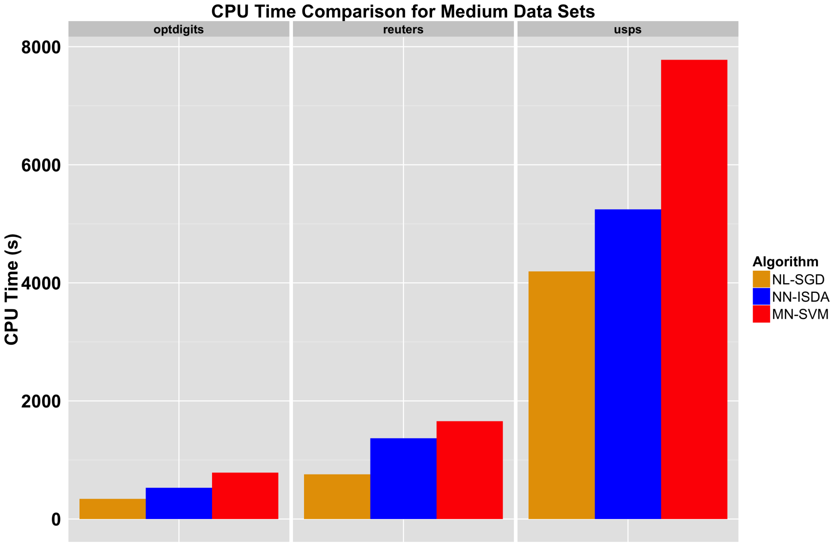

Fig. 10. CPU Time Comparison for Medium Datasets

taken into account. We tested our implementation using three different percentages of the $\boldsymbol{\alpha}$ updates, as explained in Section 3.1. These percentages are shown in the following figures on the x-axis, where 0% represents taking the last $\boldsymbol{\alpha}$ vector as the final $\boldsymbol{\alpha}^*$. For the small datasets, we ran our experiment with two and five epochs. For the medium datasets, we ran our implementation with one and two epochs. The number of epochs is depicted by the color of the bar plots, indicated by the legend. Smaller datasets would require multiple updates to the $\boldsymbol{\alpha}$ vector to ensure reaching optimality, because the algorithm loops cyclically over the data. For larger datasets, this is not the case due to their size. This is shown in Figs. 11 and 12.

Fig. 11 shows that for the smaller sized datasets, such as the *Wine*, *Glass*, and *Teach* datasets; the accuracy achieved with five epochs surpasses that with two epochs. We can also see that for the larger of the small datasets, such as *Eukaryotic* and *Prokaryotic*; the accuracy achieved with two epochs is better than using five epochs. This is because there were more updates than needed, and the minimum was

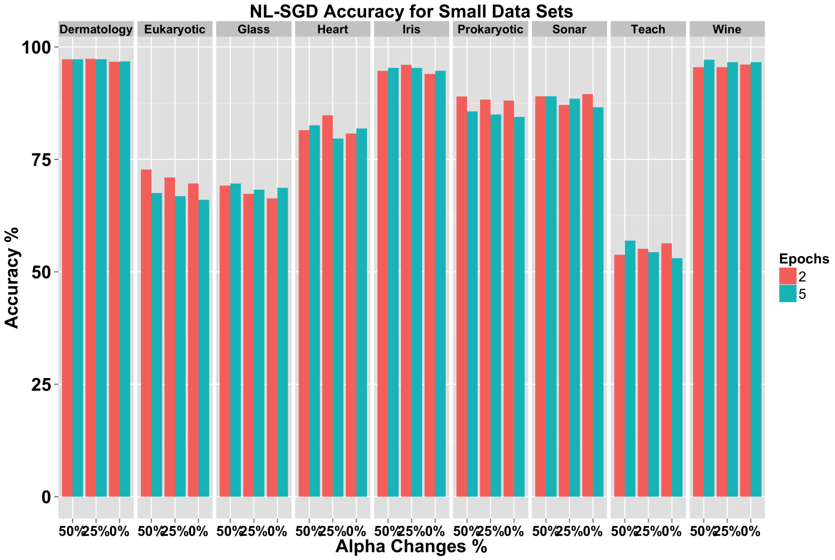

Fig. 11. NL-SGD Accuracy on Small Datasets

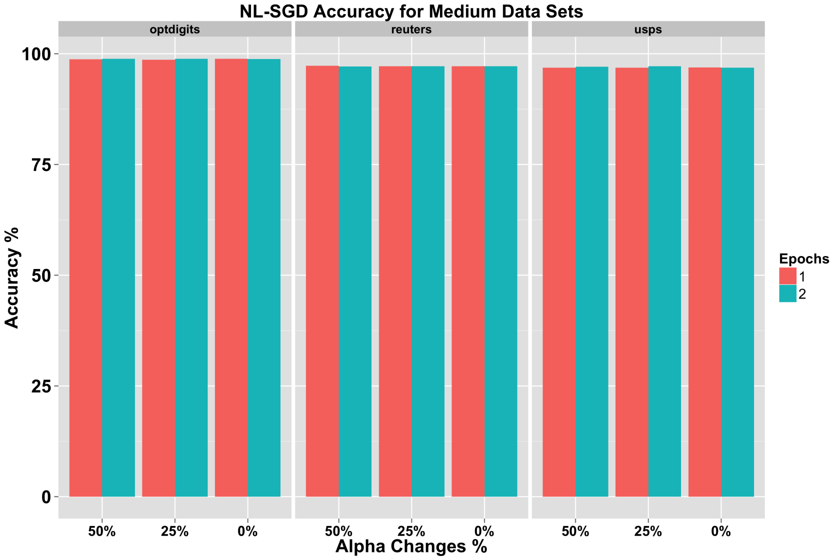

Fig. 12. NL-SGD Accuracy on Medium Datasets

overshot. We can also see that taking the average of the last 50% and 75% of alpha updates produces better results than taking the last alpha update.

From Fig. 12 we can see that there are minimal differences in the accuracy's achieved by using one or two epochs. Due to the small difference, we would recommend using one epoch to achieve faster CPU training time. Figs. 13, 14, and 15 show the CPU training times achieved by our NL-SGD implementation.

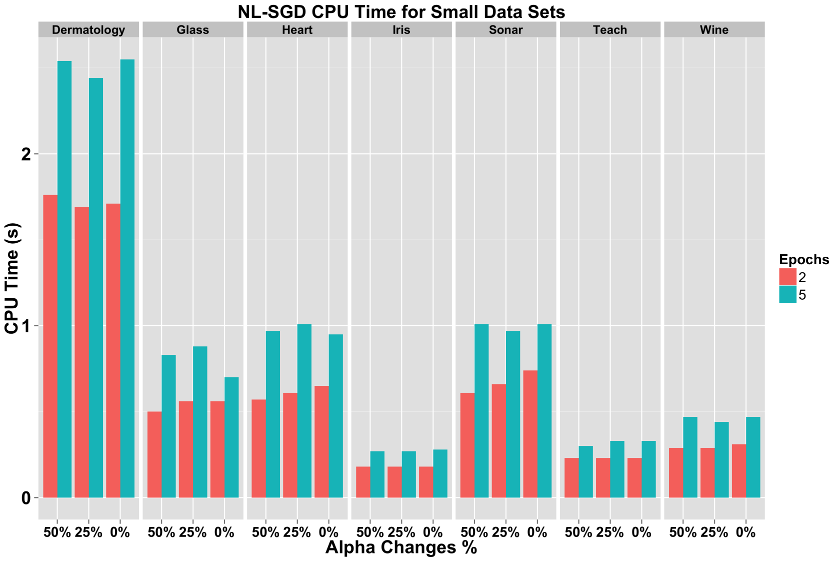

Fig. 13. NL-SGD CPU Time on Small Datasets

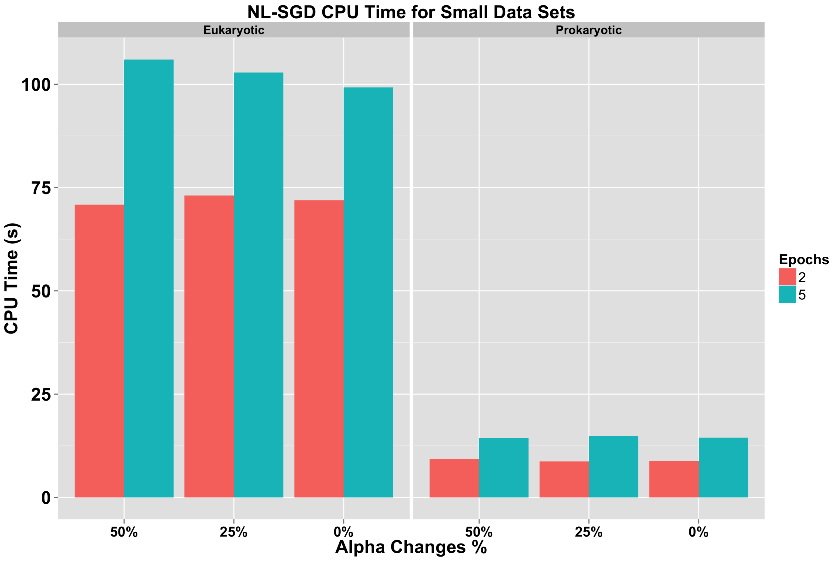

Fig. 14. NL-SGD CPU Time on Prokaryotic and Eukaryotic Datasets

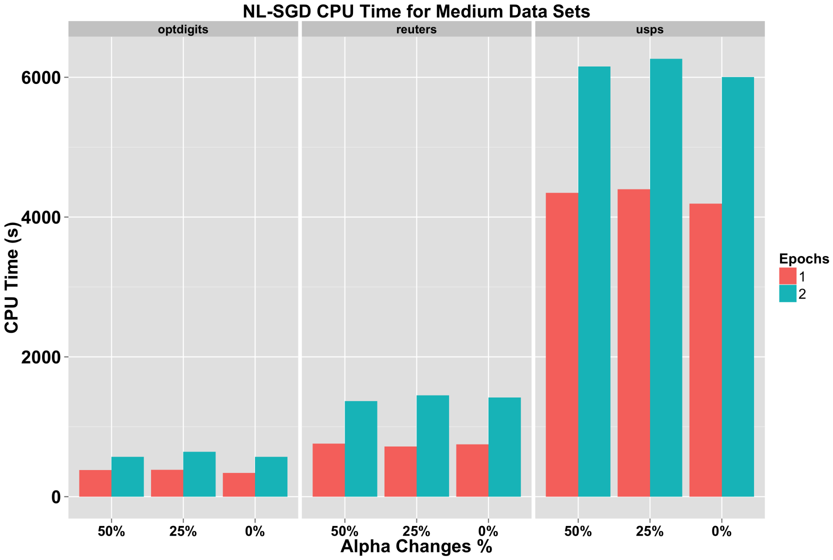

Fig. 15. NL-SGD CPU Time on Medium Datasets

We can definitely see a trade-off with balancing the number of epochs to achieve higher accuracy, while maintaining fast CPU training times. It is shown through our results that as the number of samples increases, it is preferable to use a lower epoch number. This will ensure achieving a high accuracy while maintaining low CPU training time.

4.3.3 Comparison of OL SVM with the SMO MATLAB Implementation

Table 6 shows the average error rates of the SMO and OL SVM algorithms on the test datasets given in Table 3. OL SVM is more accurate than *fitcsvm* for all but one dataset. For all the simulations, OL SVM is faster than SMO. Also note that as the dataset size increases, the speedup of OL SVM increases. It is important to note that the SMO based *fitcsvm* training time increases sharply with both a bad conditioned kernel matrix $\boldsymbol{k}$ (for example, caused by a broad Gaussian kernel or higher order polynomial) and by a large value for the penalty parameter C. Figures 17 and 16 show the typical and characteristic changes of error rates and CPU time for the two methods in 24 cross-validation runs (6 C and 4 σ values) for training a normally distributed 2 class dataset in 2-dimensional space, with 15,000 samples. The SMO algorithm has a weakness, where for large C and σ values, SMO suffers from very high CPU time, as stated previously. This is due to its iterative nature of two-coordinate descent in search for the optimal minima. With large C values, the box defined by C would be very large, thus leading to more iterations in search of the minimum value. With large σ values, the cost function would be *badly conditioned*, making the optimal value of the regularized hinge loss function, shown in Equation 3.5, difficult to find. OL SVM does not suffer from the weaknesses stated, which are linked to the SVM hyperparameters, rather it is dependent on the number of iterations which is defined by the cardinality of the dataset and the number of epochs, e.

Table 6. Average Error Rate and Speedup of OL SVM vs. SMO

Dataset	**Error Rate SMO, %**	**Error Rate OL SVM, %**	**OL SVM vs. SMO CPU Time Ratio**
Cancer	**24.00**	25.62	4.1
Sonar	34.70	**24.35**	3.68
Vote	22.68	**8.09**	1.89
Eukaryotic	36.05	**14.55**	5.67
Prototask	33.57	**11.28**	5.61
Reuters	5.64	**4.48**	4.10
Chess Board	13.62	**4.27**	7.65
Two Normally Distributed Classes	28.25	**7.82**	41.50

Figures 18 and 19 show the performances of SMO and OL SVM for the *Reuters* data. This is a very sparse dataset (only 11,069 samples in 8,315 dimensional space). The curves in all simulation have a typical sew-saw behavior due to the order of selection of C and σ. Typically, one C is selected and it is being used in combination with all the shape parameters given. The last runs are with the largest values of C and σ.

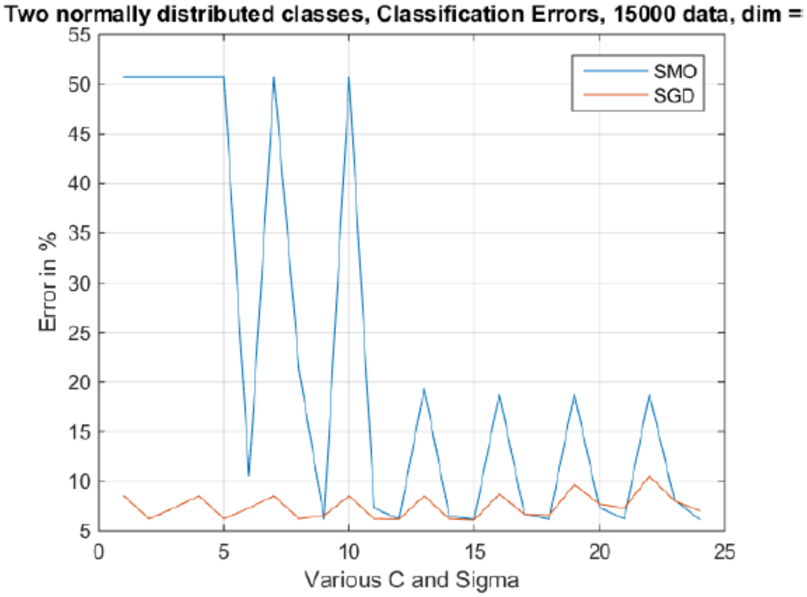

Fig. 16. Error rates of SMO and OL SVM for *two normally distributed classes* in 2-dimensional space, with a sample size of 15, 000.

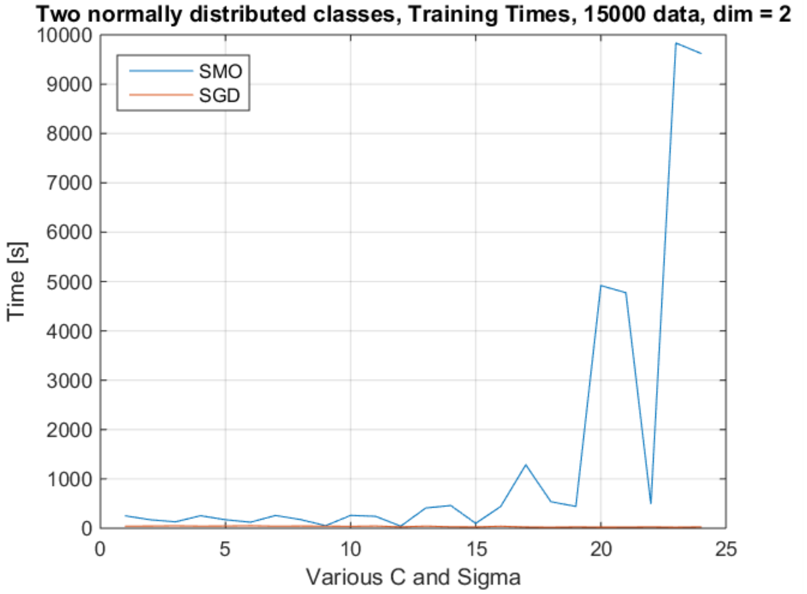

Fig. 17. CPU times of SMO and OL SVM for *two normally distributed classes* in 2-dimensional space, with a sample size of 15, 000.

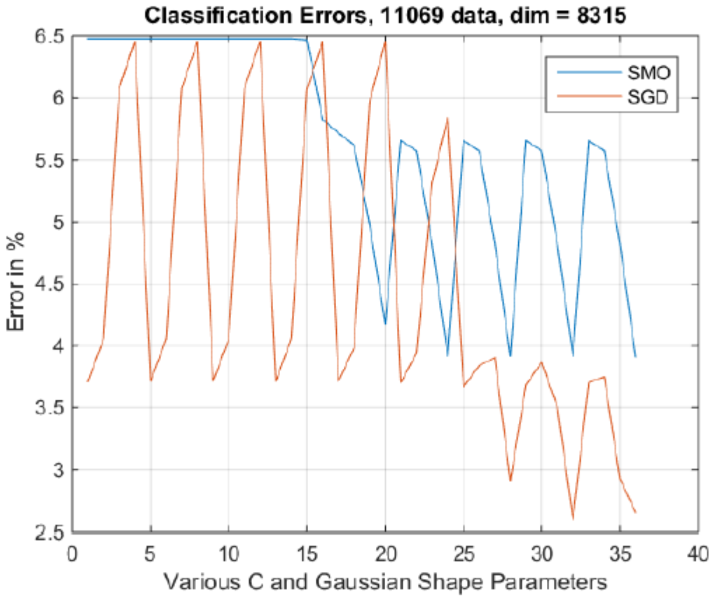

Fig. 18. CPU times of SMO and OL SVM for *two normally distributed classes* in 2-dimensional space, with a sample size of 15, 000.

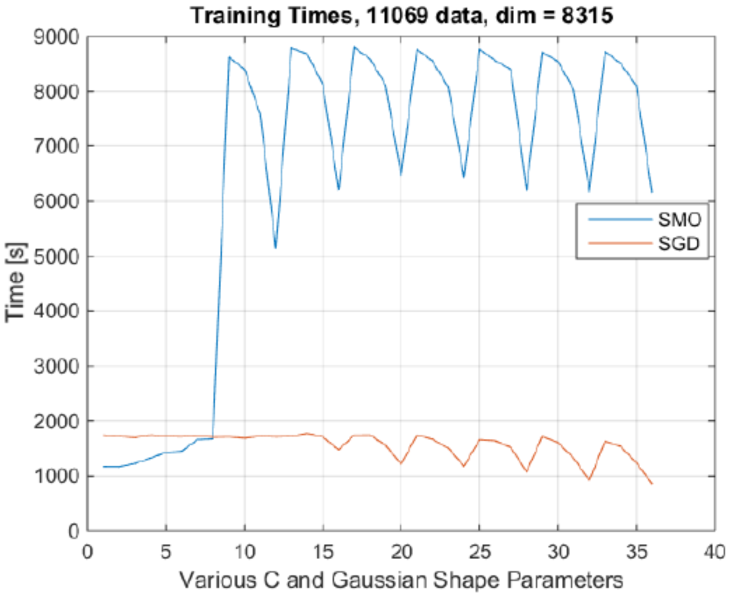

Fig. 19. CPU times of SMO and OL SVM for *two normally distributed classes* in 2-dimensional space, with a sample size of 15, 000.

More simulations and results can be found in [7] showing the superiority of OL SVM in comparison to two other stochastic methods, a modification of the algorithm Pegasos named Pegaz [19] and Norma [16, 11]. The results show that the error rate of Norma and OL SVM are similar and Pegaz has a higher error rate than both. OL SVM is 2 times faster than Norma and 4 times faster than Pegaz.

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

The potential for fast learning and building accurate models using stochastic gradient algorithms is very promising. Our implementation, named NL-SGD shown in Algorithm 3, shows CPU time speed up with respect to the fastest (known to us) non-linear SVM algorithms implemented within the publicly available GSVM code. The results shown in this thesis are indicative of the quality of the models built by NL-SGD, all the while having the best CPU training time, while keeping similar accuracies on various datasets. This thesis also introduces a new stochastic gradient based SVM learning algorithm in primal, named OL SVM. The learning classification algorithm is devised and shown in Algorithm 4 as described next. It minimizes a soft-margin, or hinge, loss without the quadratic regularization term $\mathcal{R}_{hl}$ in 3.6. This approach has some similarities with the kernel perceptron learning algorithm expressed in dual variables [7]. The experiments compare OL SVM with the classic and most popular SVM training approach, the SMO algorithm. All the experiments have been performed in MATLAB and they show the superiority of OL SVM in terms of both accuracy and CPU time needed for performing 5-fold cross-validation on various data. The comparisons have been performed on several binary classification datasets, with up to $15,000$ samples. Whether this superiority claim is right and valid for the implementation of the OL SVM algorithm on other platforms (C++, Java, Python, R) is partly answered in [14], as well as by the results presented for NL-SGD, and is still the subject of present and future investigations by using more datasets of various sizes and characteristics. An important characteristic of OL SVM and NL-SGD is that the CPU time needed for the training does not depend upon the values of penalty parameter C or on the kernel hyperparameter values (shape parameter of Gaussian kernel or the order of the polynomial kernel). This feature significantly reduces the OL SVM and NL-SGD training time for bigger values of both C and Gaussian shape parameters (i.e., for higher order polynomials) compared to the training CPU times shown by the SMO algorithm, which slows down significantly due to worse conditioning of Hessian matrix used in a dual space. The results presented in this thesis, as well as in [7], show that stochastic gradient training for the L1 SVM problem seems to be the strongest contender for classifying large datasets.

Possible future works include the following,

- Implementing OL SVM in C++ and testing both NL-SGD and OL SVM on very large datasets
- Include the ability to train models for multi-target datasets
- Parallelize both NL-SGD and OL SVM
- Implement a linear version of NL-SGD and compare its performance with OL SVM and NL-SGD

By investigating the above possible implementations, we would be expanding the range of classification and regression solutions that would be useful in various industries. The experiments conducted so far show performance improvement on small and medium datasets, and the next step would be to establish whether this will be the case for very large datasets.

Appendix A

ABBREVIATIONS

VCU	Virginia Commonwealth University
GD	Gradient Descent
SGD	Stochastic Gradient Descent
SVM	Support Vector Machine
NL SGD	Non-Linear Stochastic Gradient Descent
OL SVM	Online Support Vector Machine
MN SVM	Minimal Norm Support Vector Machine
ISDA	Iterative Single Data Algorithm
NN ISDA	Non-Negative Iterative Single Data Algorithm
SMO	Sequential Minimal Optimization
RLM	Regularized Loss Minimization
KKT	Karush-Kuhn-Tucker
RBF	Radial Basis Function
GSVM	Geometric Support Vector Machine

REFERENCES

[1] L. Bottou. *Stochastic Gradient Descent on Toy Problems.* http://leon.bottou.org/projects/sgd. 2007.

[2] C.C. Chang and C.J. Lin. *LIBSVM: A Library for Support Vector Machines.* http://www.csie.ntu.edu.tw/~cjlin/libsvm. 2011.

[3] R. Collobert and S. Bengio. “Links between Perceptrons, MLPs and SVMs”. In: *Proc. of the 21st Intl. Conf. on Machine Learning* (2004).

[4] C. Cortes and V. Vapnik. “Support-Vector Networks”. In: *Machine Learning* (1995), pp. 273–297.

[5] R. Herbrich. *Learning Kernel Classifiers - Theory and Algorithms.* The MIT press, 2002.

[6] T.M. Huang, V. Kecman, and I. Kopriva. *Kernel Based Algorithms for Mining Huge Data Sets, Supervised, Semi-Supervised, and Unsupervised Learning.* Berlin, Heidelberg: Springer-Verlag, 2006.

[7] V. Kecman and G. Melki. “Fast Online Algorithms for Support Vector Machines - Models and Experimental Studies”. In: *IEEE SoutheastCon* (2016).

[8] V. Kecman, R. Strack, and Lj. Zigic. “Big Data Mining by L2 SVMs - Geometric Insights Help”. In: *Seminar at Computer Science Department, Virginia Commonwealth University VCU* (2013).

[9] V. Kecman and Lj. Zigic. “Algorithms for Direct L2 Support Vector Machines”. In: *The IEEE International Symposium on INnovations in Intelligent SysTems and Applications, INISTA* (2014).

[10] J. Kivenin, A.J. Smola, and R.C. Williamson. “Online Learning with kernel methods”. In: *Advances in Neural Processing Systems* (2001).

[11] J. Kivinen, A.J. Smola, and R.C. Williamson. “Large Margin Classification for Drifting Targets”. In: *Neural Information Processing Systems* (2002).

[12] J.T. Kwok, I.W. Tsang, and P.M. Cheung. “Core Vector Machines: Fast SVM Training on Very Large Data Sets”. In: *Journal of Machine Learning Research* 6 (2005), pp. 363–392.

[13] *LIBSVM Data: Classification, Regression, and Multi-Label.* `http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets`. 2015.

[14] G. Melki and V. Kecman. “Speeding Up Online Training of L1 Support Vector Machines”. In: *IEEE SoutheastCon* (2016).

[15] J. Platt. “Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines”. In: *Technical Report MSR-TR-98-14* (1998).

[16] B. Schoelkopf and A. Smola. *Learning with Kernels; Support Vector Machines, Regularization, Optimization, and Beyond*. Cambridge, MA: The MIT press, 2002.

[17] S. Shalev-Shwartz and S. Ben-David. “Understanding Machine Learning From Theory to Algorithms”. In: (2014), pp. 150–166.

[18] S. Shalev-Shwartz, Y. Singer, and N. Srebo. “Pegasos: Primal estimated subgradient solver for SVM”. In: *Proc. 24th Intl. Conf. on Machine Leraning (ICML ’07)* (2007), pp. 807–814.

[19] S. Shalev-Shwartz, Y. Singer, and N. Srebo. “Pegaz: Primal estimated subgradient solver for SVM”. In: *Proc. 24th Intl. Conf. on Machine Learning (ICML07)* (2007), pp. 807–814.

[20] R. Strack. “Geometric Approach to Support Vector Machines Learning for Large Datasets”. In: *PhD dissertation, Virginia Commonwealth University* (2013).

[21] R. Strack and V. Kecman. *GitHub - GSVM*. `https://github.com/strackr/gsvm`. 2013.

[22] *UCI Machine Learning Repository: Data Sets*. `http://archive.ics.uci.edu/ml/datasets.html`. 2015.

[23] Lj. Zigic. “Novel Support Vector Machine Model and Its Algorithms Aimed at Large Datasets”. In: *PhD dissertation, Virginia Commonwealth University* (2016).

[24] Lj. Zigic, R. Strack, and V. Kecman. “L2 Support Vector Machines Revisited - Novel Direct Learning Algorithm and Some Geometric Insights”. In: *Proceedings of 19th International Conference on Soft Computing* (2013), pp. 334–339.

VITA

Gabriella Melki received her BSc degree in Computer Science from the American University of Beirut, Lebanon, in 2011. She is currently working towards her MSc. and Ph.D. degree, both in Computer Science, at Virginia Commonwealth University, Richmond, VA, USA. Gabriella's research is focused on Machine Learning, Multi-Target Learning, Optimization, and Artificial Intelligence algorithms. More specifically, she has focused on support vector machine classification and regression.

Papers:

1. **Melki G**, Kecman V, Speeding Up Online Training of L1 Support Vector Machines, IEEE SoutheastConf, Norfolk, VA, 2016
2. Kecman V, **Melki G**, Fast Online Algorithm for SVMs, IEEE SoutheastConf, Norfolk, VA, 2016

Work Experience: She worked as a Data and Programmer Analyst in the Data Warehouse and Business Intelligence department at NewMarket Corporation from 01/09/2012 - 12/12/2015.