Efficient algorithm for regularized risk minimization

Dmitry Basavin
dmbasavin@gmail.com

CTU–CMP–2013–1337
May 21, 2013

Thesis Advisor: Vojtěch Franc

The work was supported by the Grant Agency of the Czech Republic under Project P202/12/2071.

Research Reports of CMP, Czech Technical University in Prague, No. 1337, 2013

Published by
Center for Machine Perception, Department of Cybernetics
Faculty of Electrical Engineering, Czech Technical University
Technická 2, 166 27 Prague 6, Czech Republic
Efficient algorithm for regularized risk minimization

Dmitry Basavin

May 21, 2013
Declaration

I hereby declare that I have completed this thesis independently and that I have listed all the literature and publications used.
I have no objection to usage of this work in compliance with the act §60 Zákon č. 121/2000Sb. (copyright law), and with the rights connected with the copyright act including the changes in the act.

In Prague on May 21, 2013
Abstract

The recently proposed Optimized Cutting Plane Algorithm (OCA) is an efficient method for solving large-scale quadratically regularized risk minimization problems. Existing open-source library LIBOCAS implements the OCA algorithm for two important instances of such problems, namely, the Support Vector Machines algorithms for training linear two-class classifier (SVM) and for training linear multi-class classifiers (MSVM).

In this thesis we implemented an extended version of the LIBOCAS library which is able to solve the risk minimization problems with a more generic risk function. In particular, our solver allows the risk to be a generic piece-wise linear function. We give necessary mathematical background of the OCA algorithm and we describe details of our implementation. We show how to use our generic library to implement solvers for the SVM and the MSVM algorithms. We experimentally compare our implementation with the LIBOCAS on several benchmark data sets. The comparison shows that our library obtains exactly the same solution as the LIBOCAS requiring a comparable convergence time while being able to deal with far more generic risk functions.

Abstrakt

ceskou verzi jsem necetl


Cílem této práce je naimplementovat rozšířenou verzi knihovny LIBOCAS, která bude schopna řešit minimalizační problémy s obecnější rizikovou funkcí. Nejprve zopakujeme matematické základy algoritmu OCA. Potom se zmiňujeme o implementačních detailech naší knihovny. Nakonec využijeme naší knihovnu pro vyřešení několika instancí BSVM a MSVM problémů. Výsledky experimentů ukazují, že naše knihovna získává přesně stejné řešení jako knihovna LIBOCAS a má srovnatelný čas konvergence, i když je schopná se vypořádat s mnohem obecnější rizikovou funkcí.
Contents

1 Introduction 2

2 State of the art 4
   2.1 Dedicated solvers 4
   2.2 Generic solvers 4
      2.2.1 Approximate solvers 5
      2.2.2 Exact solvers 5

3 Algorithm 7
   3.1 Approximating the risk $R$ 8
   3.2 Solving the reduced problem 9
   3.3 Line-search 9
   3.4 Adding a new cutting plane 10
   3.5 Stopping conditions 11

4 Implementation 12
   4.1 Overview 12
   4.2 Data representation 13
   4.3 Solver representation 14

5 User’s guide 15
   5.1 C++ way 15
   5.2 MEX interface 17

6 Applications 19
   6.1 Linear Two-class SVM classifiers 19
   6.2 Linear Multi-class SVM classifiers 20

7 Experiments 21
   7.1 Data sets 21
   7.2 Benchmark results 22
   7.3 Discussion 24

8 Conclusions 25

Bibliography 26
1 Introduction

Many machine learning methods can be formulated as the unconstrained regularized risk minimization problem. Many machine learning algorithms can be formulated as a convex risk minimization problem. In particular, a large class of such algorithms are translated to the following unconstrained problem:

$$\mathbf{w}^* = \arg\min_{\mathbf{w} \in \mathbb{R}^n} P(\mathbf{w}) := \left[ \frac{\lambda}{2} \| \mathbf{w} \|^2 + R(\mathbf{w}) \right]$$  \hspace{1cm} (1.1)

where $\mathbf{w} \in \mathbb{R}^n$ is a parameter vector we are trying to learn, $\lambda \in \mathbb{R}^{++}$ is a fixed positive constant, $\frac{1}{2}\| \mathbf{w} \|^2$ is a quadratic regularization term which restricts the learned parameter vector, and $R(\mathbf{w})$ is a risk function evaluating fit of the parameter vector to the training examples that tells us how well the learned parameter vector "explains" the training data points. The risk function is usually non-differentiable and complex, and hence difficult to deal with directly.

Globalni poznamka: Navrhuju pouzivat “training examples” namisto “training points”.

In this thesis we implement a generic and efficient algorithm proposed in [17] for solving a special case of the problem (1.1). In this thesis we implement the Optimized Cutting plane Algorithm (OCA) [17] which is a specialized solver for large-scale instances of the problem (1.1). In our implementation we constrained ourselves to the problems where the risk function is of the form:

$$R(\mathbf{w}) = \max \left\{ 0, \sum_{i=1}^{m} \max_{j \in \{1,...,k\}} \left( \langle \mathbf{a}_{ij}, \mathbf{w} \rangle + b_{ij} \right) \right\}, \hspace{1cm} (1.2)$$

where $\mathbf{a}_{ij} \in \mathbb{R}^n$, $b_{ij} \in \mathbb{R}$, $\forall i, j$ are the fixed parameters. This function is convex and always non-negative. The risk is composed of a sum of piece-wise linear functions. It is seen that the risk is convex and non-negative. In the sequel, we denote the problem (1.1) with the risk (1.2) as the Quadratically Regularized Piece-Wise Linear (QR-PWL) minimization problem.

Dale v textu pouzivejte zkratu QR-PWL misto odkazovani (1.1) a (1.2).

Multiple prominent machine learning algorithms can be viewed as instances of the problem (1.1) with the risk function (1.2), including Linear Support Vector Machines [4], Maximal Margin Structured Output Classification [16], Novelty Detection [12], Gaussian Processes [18], Logistic Regression [3], Conditional Random Fields [11], Lasso [15], Ordinal Regression [6] and many others.

Despite the fact that the problem (1.1) with the risk function (1.2) has so many applications in machine learning, the off-the-shelf solvers are often found impractical in the real-world situations. Nerozumim smyslu teto vety. Therefore, we believe that implementing the efficient (but still very generic) algorithm proposed in [17] will benefit many researchers in the machine learning community.
It will especially help the scientists who are working on prototyping the new large-scale optimization algorithms. The established practice was that after invention of a new learning algorithm the researches spent time on developing an efficient solver tailored for the particular optimization problem. The specialized algorithms are often order of magnitude faster than of-the-shelf-solvers, however, their extension to other optimization problems is typically complicated. Our aim is to implement a generic solver whose efficiency would be comparable to specialized algorithms. We hope that our solver will become a useful tool for developing new large-scale learning methods.

The rest of this document is structured as follows. Section 2 talks about the state-of-the-art algorithms solving different instances of problem (1.1). Section 3 presents the OCA algorithm as applied to the unconstrained risk minimization problems with the risk defined as in (1.2). In particular, it shows the pseudocode of the algorithm and points out the relevant details about every step of the pseudocode. In Section 4, we discuss the implementation details of our solver. In section 5, we provide a user’s guide for our library with examples showing how to use the solver in C++ applications and in Matlab. After that, we provide a user’s guide (Section 5), where readers can find out which steps are necessary to launch our solver in both C++ and MATLAB environments. Then in Section 6 we will talk about the applications of our algorithm. In particular, we will show how to use our library to implement a solver for several well-known machine learning problems. In section 6, we describe two important applications, namely, we show how to apply the implemented algorithm for learning two-class SVM classifiers and multi-class SVM classifier. Finally, we test our algorithm on several data sets and compare the results with the state-of-the-art solver LIBOCAS (Section 7). In section 7, we experimentally compare our implementation with the LIBOCAS library on several benchmark data sets. Section 8 talks about the future work and concludes the thesis. Finally, section 8 gives conclusions and outlines the future work.
2 State of the art

There are various methods one can choose to optimize the problem (1.1). In this section, we will briefly describe several such methods and point out their pros and cons. A large effort has been put by the ML community to development of solver for particular instances of the problem (1.1). In this chapter we outline main categories of existing solvers.

2.1 Dedicated solvers

Traditionally, dedicated solvers have been developed to solve the instances of (1.1) with particular risk functions. For example, to solve an SVM classification problem, one can choose from a plethora of options: SVM-light [8], LIBSVM [2], SVM-perf [9], SVMlin [13], and many others.

Dedicated solvers have been proposed for several important instances of the problem (1.1) with particular form of the risk function. Probably the most investigated is the optimization problem which has to be solved when learning two-class SVM classifiers (see Section XX). Several optimization strategies are implemented in popular SVM packages like SVM-light [8], LIBSVM [2], SVM-perf [9] or SVMlin [13] to new a few.

The disadvantage of the dedicated solvers is that it might be very difficult to create them. This difficulty stems from the fact that an efficient implementation requires a deep understanding of a problem at hand. Also, it is usually a very time-consuming task. As a result, there are plenty of important instances of problem (1.1) that do not have an existing dedicated solver.

The dedicated algorithms are often order of magnitudes faster than the of-the-shelf solvers. The efficiency is obtained by exploiting particular structure of the optimization problem. On the other hand, extension of such algorithms to solve other (even slightly modified) optimization problems is typically complicated if not impossible.

2.2 Generic solvers

Generic solvers do not require any specific information about the risk function used in (1.1). Hence, one can apply those solvers to optimize various instances of problem (1.1). Unfortunately, that usually negatively impacts their performance on the big datasets, because they are not able to leverage the special structure of the risk function to optimize the problem in a most efficient way.

In this discussion we will restrict ourselves to the subset of the generic solvers which utilize a first-order oracle to access the objective function of problem (1.1). The first-order oracle can be viewed as a black box. In every iteration of the algorithm, it receives the current solution as a query point and returns the value
2.2 Generic solvers

of the objective function and a subgradient of the objective function at that query point.

First note that the QP-PWL problem can be easily converted to an equivalent convex quadratic program for which many off-the-shelf solvers exists. However, the instance size (i.e. the number of variables and the number of linear constraints) of the equivalent QP problems which appear in real-life ML applications rules out usage of the generic QP solvers (e.g. the computational time of the Interior Point Methods scales at least quadratically with the number of variables/constrains).

There exist specialized solvers which can optimize the problem (1.1) with almost arbitrary risk function. The generic solvers typically access the risk function indirectly via the first-order oracle, i.e. the oracle returns function value of the risk and its sub-gradient evaluate at a given parameter vector.

Next, we will discuss the two types of the first-order oracle generic solvers: approximate and exact ones.

In the next sections we will discuss two classes of the generic solvers. First, the approximate on-line solvers and, second, precise batch solvers delivering solution with a certificate of optimality.

2.2.1 Approximate solvers

Since solving the problem (1.1) to optimality is generally a hard task, different approximate algorithms have been created. Stochastic Gradient Descent algorithm (SGD) [19] is a famous representative of the approximate solvers. In each iteration of SGD, the gradient is estimated based on a single point randomly picked from a dataset. Despite its simplicity it is well-known that the SGD algorithm almost surely converges under mild conditions (see e.g. [?]).

In practise, the stochastic algorithms usually converge faster than the exact methods. The disadvantage of such methods is that the stochastic process has no stopping condition. Hence, there is no way to ensure that the obtained solution is close to optimal.

In practise the stochastic on-line methods converge faster at the first stages of the optimization process but stall in the vicinity of the optimum. A considerable disadvantage of the on-line methods is lack of theoretically sound stopping conditions.

2.2.2 Exact solvers

Exact algorithms always produce an $\epsilon$-optimal solution for prescribed $\epsilon > 0$. They also provide an easy way to verify that the particular solution is indeed $\epsilon$-optimal (certificate of optimality).

The exact solvers provide a certificate of optimality. The exact solvers guarantee that the found solution does not differ from the optimal one more than a prescribed $\epsilon$. The difference is typically measured in terms of the objective value. Such solution is denoted as the $\epsilon$-optimal solution.

Bundle methods [?] are an important example of such algorithms. Bundle methods are an important example of generic solvers applicable for convex risk minimization coming with optimality guarantees. The bundle methods are an
improved version of the Cutting Plane Algorithm (CPA) [10]. In particular, a proximity control function is introduced. That function prevents the unstable "zig-zaging" behaviour of CPA, and therefore speeds up the coverage. In particular, an auxiliary prox-term is introduced to the objective function in order to remove the "zig-zag" behaviour of the plain CPA.

The modification of the bundle methods was proposed by Teo in [14] and is called Bundle Method for Risk Minimization (BMRM). A modified version of the bundle methods, called the Bundle Method for Risk Minimization', was proposed in [14]. The BMRM algorithm leverages the specific structure of the problem (1.1) to optimize it efficiently. In particular (as explained in [?]), only the risk term \( R(w) \) is approximated by the cutting-plane model while the regularization term \( \frac{1}{2} ||w||^2 \) is used to stabilize the optimization, i.e. it serves as an natural prox-term known from the original bundle methods. The resulting algorithm is highly modular and it was proven to converge in \( O\left(\frac{1}{\epsilon}\right) \) iterations to an \( \epsilon \)-precise solution.

The BMRM algorithm can be drastically accelerated whenever an efficient line-search algorithm for the objective function of the problem (1.1) is available. An accelerated variant of the BMRM for solving linear SVM problem was first proposed in [?]. Later, in [17] the method was generalized for solving (1.1) with an arbitrary risk \( R(w) \) for which the line-search procedure exists. Finally, a fully generic version of the algorithm was proposed in [14], which puts no restrictions neither on a risk function, nor on a regularizer. To není upne pravda. Omezení jsou: regularizator musí byt hladká funkce a line-search musí existovat.

There are several existing implementations of the BMRM and its accelerated versions, e.g. the SHOGUN machine learning toolbox [?] and the open-source package LIBOCAS [?] provide such implementations. The LIBOCAS library implements the algorithm proposed in [17], but does it so for two particular risk functions only. Namely, the implementation is provided for learning the two-class linear SVM classifiers and the Multi-class linear SVM classifiers. Therefore, our effort in this thesis can be viewed as an attempt to extend the LIBOCAS library to be able to handle the optimization problems with a more general risk function, since both Two-class and Multi-class SVM risk functions are the particular instances of our risk function defined in (1.2). The goal of this thesis is to extend the LIBOCAS library so that the new implementation is able to handle the optimization problems with a more general risk function, namely, any piece-wise linear function can be used.
3 Algorithm

The algorithm that we have implemented during the course of this thesis was described in detail in [17]. A detailed description of the OCA algorithm implemented in this thesis is provided in [17]. Here, we provided only a brief description of the algorithm necessary to explain our implementation. According to its authors, the OCA algorithm extends the BMRM algorithm in two ways. First, unlike the BMRM, it uses the solution of the reduced problem as a direction in the line-search to directly minimize the original problem (1.1). Second, the OCA simultaneously optimizes the original objective and its cutting plane approximation (called the reduced problem) while the BMRM does not care of the original objective at all. The original objective is optimized by a line-search procedure which can be implemented very efficiently for the piece-wise linear risk functions. Second, it introduces the new cutting plane selection strategy that reduces the number of cutting planes required for convergence. Second, the OCA introduces a new cutting plane selection strategy which attempts to approximate the objective function in a vicinity of the optimal solution by which it reduces the number of iterations. Algorithm 1 shows a pseudo-code of the OCA algorithm.

Algorithm 1 Optimized Cutting Plane Algorithm (OCA)

1: $t \leftarrow 0$ and $w_0^b \leftarrow 0$
2: Add an initial cutting plane: compute $s_0$ and $r_0$ at the point $w_0^b$.
3: repeat
4: $t \leftarrow t + 1$
5: Compute $w_t$ by solving the reduced problem. (Sec. 3.2)
6: Compute $w_t^b$ using vectors $w_{t-1}^b$ and $w_t$. (Sec. 3.3)
7: Compute $w_t^c$ using $w_t$ and $w_t^b$. (Sec. 3.4)
8: Add a new cutting plane: compute $s_t$ and $r_t$ at point $w_t^c$. (Sec. 3.4)
9: until best-so far solution $w_t^b$ is good enough

As we can see from the presented pseudocode, the main idea of the algorithm is the following. In every iteration, we compute a new cutting plane by solving the reduced problem and applying the line-search procedure. Every additional cutting plane helps us better approximate the risk function $R$ of problem (1.1). As the approximation becomes better, we get a tighter lower bound on the objective value of problem (1.1). When the gap between the value of the approximated risk and the actual risk goes below the particular threshold, we proclaim that the obtained solution is good enough and terminate the algorithm.

The OCA algorithm iteratively solves the reduced problem (step 5) and uses
the reduced problem solution to i) improve the primal objective by line-search procedure (step 6) and the ii) to construct a new cutting plane in the vicinity of the best so far solution (step 7 and 8). A new single cutting plane is added in each iteration making the reduced problem objective progressively tighter approximation of the original objective. The algorithm stops if the gap between the original objective and the reduced problem objective drops below a prescribed \( \epsilon > 0 \).

In the rest of this section, we will briefly review the method as it is proposed in [17]. Also, we will provide some additional details about the main steps of the procedure, which are required to solve the instances of problem (1.1) with risk (1.2). In the rest of the section, we briefly describe individual steps of the OCA algorithm. We give formulas necessary for implementation of the OCA algorithm for solving the QR-PWL problem. In particular, we will show how to compute the subgradient of risk (1.2) and how to solve the line-search problem. Finally, we will talk about the stopping conditions and the convergence guarantees of the OCA algorithm. Finally, we discuss variants of the stopping conditions used in our implementation.

### 3.1 Approximating the risk \( R \)

In the established terminology the problem (1.1) is denoted as the master problem. Let us define the reduced problem of (1.1) as follows:

\[
\mathbf{w}_t^* = \arg\min_{\mathbf{w} \in \mathbb{R}^n} P_t(\mathbf{w}) := \left[ \frac{\lambda}{2} \|\mathbf{w}\|^2 + R_t(\mathbf{w}) \right]. \tag{3.1}
\]

Next, we derive a piece-wise linear approximation \( R_t(\mathbf{w}) \) of risk function \( R(\mathbf{w}) \). As we have already seen in the previous section, the risk defined in (1.2) is convex, hence it can be approximated at any point \( \mathbf{w}' \) by a linear underestimator

\[
R(\mathbf{w}) \geq R(\mathbf{w}') + \langle s', \mathbf{w} - \mathbf{w}' \rangle, \quad \forall \mathbf{w} \in \mathbb{R}^n, \tag{3.2}
\]

where \( s' \in \partial R(\mathbf{w}') \) denotes a subgradient of \( R \) at point \( \mathbf{w}' \). Later in this document we will talk about how to compute the subgradient \( s' \) when the risk has the form defined in (1.2). Formulas for computing the sub-gradient of the PWL risk (1.2) are discussed in Section XXX. By using shorthand \( r' = R(\mathbf{w}') - \langle s', \mathbf{w}' \rangle \) we can rewrite (3.2) as

\[
R(\mathbf{w}) \geq \langle s', \mathbf{w} \rangle + r'. \tag{3.3}
\]

The set of points \( \{ \mathbf{w} \in \mathbb{R}^n \mid \langle s', \mathbf{w} \rangle + r' = 0 \} \) is called a cutting plane.

A better approximation of the risk \( R \) can be achieved. Instead of using a single cutting plane, we can compute a collection of cutting planes \( \{ \langle s_i, \mathbf{w} \rangle + r_i = 0 \mid i = 1, \ldots, t \} \) at \( t \) distinct points \( \{ \mathbf{w}_1, \ldots, \mathbf{w}_t \} \) and take their maximum. A cutting plane approximation of the risk \( R \) can be obtained by computing a collection of cutting planes a distinct points \( \{ \mathbf{w}_1, \ldots, \mathbf{w}_t \} \) and taking their piece-wise maximum:

\[
R_t(\mathbf{w}) = \max_{i \in \{1, \ldots, t\}} \left( \langle s_i, \mathbf{w} \rangle + r_i \right). \tag{3.4}
\]
3.2 Solving the reduced problem

The zero cutting plane is added to the formula because our risk (1.2) is non-negative by assumption. The subscript $t$ denotes the number of cutting planes used in the approximation $R_t$. It directly follows from (3.2) that $R_t$ is a lower bound of $R$ tight at points $\{w_1, \ldots, w_t\}$. In turn, the reduced problem objective $P_t$ is a lower bound of master problem objective $P$.

3.2 Solving the reduced problem

By substituting (3.4) into (3.1) we can rewrite the reduced problem as an equivalent quadratic program

\[
\begin{align*}
(w_t, \xi_t) := \min_{w \in \mathbb{R}^n, \xi \in \mathbb{R}} & \quad \frac{\lambda}{2} \|w\|^2 + \xi \\
\text{subject to} & \quad \xi \geq \langle s_i, w \rangle + r_i, \quad i = 1, \ldots, t \\
& \quad \xi \geq 0.
\end{align*}
\]

(3.5)

The number of constraints in (3.5) is $t + 1$, where $t$ is the number of the cutting planes. As this number is typically much smaller than the dimension of the parameter vector $n$, it is more efficient to solve the reduced problem (3.5) by optimizing its (Lagrange) dual formulation

\[
\alpha_t := \min_{\alpha \in \mathbb{R}^n} \left[ \sum_{i=1}^{t} \alpha_i r_i - \frac{1}{2\lambda} \|s_i \alpha_i \|^2 \right] \\
\text{subject to} \quad \sum_{i=1}^{t} \alpha_i \leq 1 \\
& \quad \alpha_i \geq 0, \quad i = 1, \ldots, t.
\]

(3.6)

The dual reduced problem (3.6) can be efficiently solved by the standard QP solvers. In our implementation we used the LIBQP library\(^1\). After obtaining the dual solution $\alpha_t$, the primal solution $w_t$ is recovered by

\[
w_t = - \sum_{i=1}^{t} s_i [\alpha_t]_i
\]

(3.7)

3.3 Line-search

After calculating the solution of the reduced problem $w_t$, the OCA algorithm finds the new best-so-far solution $w_t^b$ by searching along a line starting at the previous best-so-far solution $w_{t-1}^b$ and crossing the current reduced problem solution $w_t$. That is, the OCA algorithm executes the following line-search procedure

\[
w_t^b = w_{t-1}^b (1 - x_t) + w_t x_t,
\]

(3.8)

where

\[
x_t = \arg\min_{x \geq 0} P(w_{t-1}^b (1 - x) + w_t x).
\]

(3.9)

\(^1\)URL of LIBQP
After we plug in the definitions of \( P \) and \( R \) from (1.1) and (1.2), respectively, we can rewrite (3.9) as follows:

\[
x_t = \arg\min_{x \geq 0} \left[ \frac{1}{2} k_0 x^2 + k_1 x + \max \left\{ 0, \sum_{j=1}^{m} \max_{j'=1,\ldots,k} \left( A_{j'i} x + B_{j'i} \right) \right\} \right], \tag{3.10}
\]

where

\[
\begin{align*}
k_0 &= \lambda ||w_{t-1}^b - w_t||^2 \\
k_1 &= \lambda \langle w_{t-1}^b, w_t - w_{t-1}^b \rangle \\
A_{j'i} &= \langle a_{ij}, w_t - w_{t-1}^b \rangle \\
B_{j'i} &= \langle a_{ij}, w_{t-1}^b \rangle + b_{ij}
\end{align*}
\]

and \( a_{ij} \in \mathbb{R}^n \) and \( b_{ij} \in \mathbb{R} \), \( \forall i, j \) are the same as in (1.1). It is seen that (3.10) is an univariate convex problem. In our implementation, we use an algorithm described in [17] which solves the line-search problem (3.10) exactly in \( O(mk^2 + mk \log(mk)) \) time. Note that a faster algorithm has been proposed in [7].

### 3.4 Adding a new cutting plane

To compute a new cutting plane we first need to obtain the point \( w_t^c \) which lies in a vicinity of the best-so-far solution \( w_t^b \) defined by (3.8). In particular, the point \( w_t^c \) is computed as

\[
w_t^c = w_t^b(1 - \mu) + w_t \mu \tag{3.12}
\]

where \( \mu \in (0, 1] \) is a fixed parameter. It was experimentally shown in [17] that setting \( \mu = 0.1 \) works consistently well and tuning \( \mu \) for a particular data brings a little improvement.

The next step is to compute the subgradient \( s_{t+1} \in \mathbb{R}^n \). In our algorithm we use the index \( t \) namisto \( t+1 \). Je treba byt konzistentni. First, we construct a vector \( j^* = (j_1^*, j_2^*, \ldots, j_m^*) \in \mathbb{R}^m \) such that

\[
j_i^* = \arg\max_{j=1,\ldots,k} \left( \langle a_{ij}, w_t^c \rangle + b_{ij} \right), \quad \text{for } i = 1, \ldots, m, \tag{3.13}
\]

where \( a_{ij} \in \mathbb{R}^n \) and \( b_{ij} \in \mathbb{R} \), \( \forall i, j \) are the same as in (1.1). Then we compute

\[
s_{t+1} = \sum_{i=1}^{m} a_{ij_i^*} \tag{3.14}
\]

After obtaining the subgradient \( s_{t+1} \), the next step is to compute \( r_{t+1} \in \mathbb{R} \) as follows:

\[
r_{t+1} = R(w_t^c) - \langle s_{t+1}, w_t^c \rangle. \tag{3.15}
\]

Having \( s_{t+1} \) and \( r_{t+1} \), the new cutting plane is defined as

\[
\{ w \in \mathbb{R}^n \mid \langle s_{t+1}, w \rangle + r_{t+1} = 0 \}. \tag{3.16}
\]
3.5 Stopping conditions

The process of solving the reduced problem, running the line-search procedure and adding a new cutting plane is going to last until the stopping criterion is met. The iterative process defined by the OCA algorithm is executed until the current best-so-far solution $w^b_t$ is guaranteed to be in the vicinity of the optimal solution $w^*$. At each iteration the OCA algorithm provides an upper bound $P(w^b_t)$ and the lower bound $P_l(w_t)$ of the (unknown) optimal solution $P(w^*)$.

We stop the execution of the OCA algorithm when the following inequality holds:

$$P(w^b_t) - P_l(w^b_t) \leq \epsilon P(w^b_t) \quad (3.17)$$

where $\epsilon$ is a prescribed parameter. It can be easily shown that the condition (3.17) is equivalent to

$$P(w^b_t) \leq (1 + \epsilon) P(w^*) \quad (3.18)$$

Instead of using the relative deviation from the optimal value, one can also use the absolute deviation

$$P(w^b_t) - P_l(w^b_t) \leq \epsilon \quad (3.19)$$

equivalent to

$$P(w^b_t) \leq \epsilon + P(w^*) .$$

It is shown in [17] that the maximal number of iterations the OCA algorithm requires to satisfy the stopping condition (3.19) scales with $O\left(\frac{1}{\epsilon}\right)$.
4 Implementation

In this section we will discuss the implementation details of our algorithm. In particular, we will show the architectural overview of the library and highlight the main ideas behind its design. Then we will present some essential information about the main classes and interfaces that contain the public API methods provided to the end-users.

4.1 Overview

As we have already seen in Section 1, a wide variety of machine learning problems can be solved using our algorithm. Hence, one of the main architectural challenges was the following. We wanted to separate the implementation parts which would differ from problem to problem, from those which would remain the same. In that case, we could make our library modular, reusable and flexible.

Figure 4.1 shows the class diagram of the core part of the library. This figure conforms to the UML class diagram notation. The explanations of the used symbols can be found in the multiple references on the Internet, or in the Martin Fowler’s book [5].

![Figure 4.1 Class diagram of library core.](image)

As we can see from the Figure 4.1, there are two main entities: the PWLSolver class and the PWLData interface. In a nutshell, the PWLSolver class encapsulates the main computational logic of the algorithm. It is the part of the library which is meant to be untached by the user no matter what particular instance of the risk function (1.2) the problem has. On the other hand, the PWLData
4.2 Data representation

interface represents the varying part of the algorithm. In particular, it includes all the methods which work with the instance data directly.

Therefore, if an end-user decides to optimize the problem (1.1) with any variation of the risk function (1.2), then all she needs to do is to provide her own implementation of the PWLData interface. And the rest of the library can remain untouched.

4.2 Data representation

Every optimization problem has the instance data associated with it. In our case, it is the vectors \( a_{ij} \in \mathbb{R}^{n} \forall i, j \) and scalars \( b_{ij} \in \mathbb{R} \forall i, j \), which are used in the definition of the risk function (1.2). which define an instance of the PWL risk (1.2)

The PWLData interface is displayed in the top right corner of Figure 4.1. There we can see the overview of its internal structure. First, it exposes three publicly accessible integers, \( m, n, k \), which represent the corresponding dimensions of the instance data. Second, it declares all the methods that work directly with the instance data and are required by the main algorithm. In particular, there are two such methods: ComputeAffine and ComputeSubgradient.

ComputeAffine

Several times during the execution of the algorithm we need to evaluate the expression \( \langle a_{ij}, w \rangle + b_{ij} \), where \( a_{ij} \in \mathbb{R}^{n} \) and \( b_{ij} \in \mathbb{R} \forall i, j \) are the fixed instance data and \( w \) is a variable vector.

That is what ComputeAffine function is designed to do: it takes a vector \( w \) as an input parameter and returns a matrix \( C \in \mathbb{R}^{m \times k} \) such that

\[
C_{ij} = \langle a_{ij}, w \rangle + b_{ij}
\]

for all \( i = 1, \ldots, m \) and \( j = 1, \ldots, k \).

ComputeSubgradient

One of the main steps of the OCA algorithm is adding the new cutting planes. To do that, we eventually have to compute the subgradient \( s \) of the risk function \( R \) as follows:

\[
s = \sum_{i=1}^{m} a_{ij}^*\]

where \( j^* \in \mathbb{R}^{m} \) is a given vector of indices and \( a_{ij} \in \mathbb{R}^{n} \forall i, j \) are fixed instance data (please refer to the Section 3.4 of this document for further details).

And that’s exactly what ComputeSubgradient does: it takes a vector \( j^* \) as an input parameter and returns a vector representing the subgradient \( s \).
4 Implementation

Implementing the PWLData interface

At the bottom row of the Figure 4.1 there are three entities that we have not mentioned yet. Those are the three existing implementations of the PWLData interface: one is for the Two-class SVM, another one is for the Multiclass SVM and the last one is for the interfacing with the MATLAB environment. Each of those implementations knows how to most effectively initialize, store and process the data using the domain-specific knowledge (please refer to the Section 6 for further details).

To deal with any other variation of the risk function (1.2) used in problem (1.1), the user simply has to provide the problem-specific implementation of the PWLData interface. In particular, that means implementing two data manipulation functions ComputeAffine and ComputeSubgradient, that we have discusses earlier. It may also include writing the initialization and the clean up code, for example.

4.3 Solver representation

In the top left corner of the Figure 4.1 there is the PWLSolver class. As we have mentioned before, this class encapsulates the main computational logic of the algorithm.

Upon creation, it receives a reference on an instance of the PWLData interface (see Section 4.2). That is, if our algorithm wants to do any kind of operations involving the instance data, it has to do so via a public call on the PWLData object. As a result, the PWLSolver class is completely separated from the instance data. This means that it does not know how the instance data is initialized, stored or dealt with, which makes the algorithm completely generic. It also means that the end-user, no matter what particular variation of the risk function (1.2) she is dealing with, does not need to change the source code of the PWLSolver to make everything work.

Upon completion, the PWLSolver will output the solution vector $w$ and a series of book-keeping flags and data. That includes the values of the master and the reduced problems after each iteration of the algorithm, the total time and the total number of iterations, etc. Further details about the setup steps and the output of the algorithm can be found in the User’s Guide (Section 5).
5 User’s guide

Let us consider the situation when someone decides to use our library to solve the instance of problem (1.1) with risk function (1.2). There are two ways one can do that. The first way is using the C++ classes to implement the PWLData interface and calling the solver directly from the C++ code. The other way is to call the solver from MATLAB environment via the mex interface. In this section, we will describe both possibilities. We will present the step-by-step setup instructions along with the code snippets, and talk about the expected output. As an illustration, we will be optimizing the following problem:

$$w^* = \arg\min_{w \in \mathbb{R}} \left[ \frac{1}{2} w^2 + \max\{-w + 4, w + 2\} \right].$$

(5.1)

Note that this is indeed the instance of problem (1.1) with risk function (1.2). Here $m = 1$, $n = 1$, $k = 2$, $\lambda = 1$ and $a_{11} = -1$, $a_{12} = 1$, $b_{11} = 4$, $b_{12} = 2$. Also note that the risk term is always non-negative.

5.1 C++ way

The core of our library is written in C++ to achieve a better performance. If the speed of convergence is crucial for the end-user, then she might want to write her code in C++ as well, in order to be able to interact with our solver directly. Otherwise, we suggest to take a look at the mex interface, which is a more user-friendly way to perform the computation (please consult the Section 5.2 for further details).

The first step is to write the implementation of the PWLData interface, which encapsulates the instance data of problem (5.1).

```
#include "data.h"

class MyPWLDataImpl: public PWLData {
public:

    void ComputeAffine(double* w, double** C) {
        /* C[i][j] = a[i][j] * w + b[i][j] */
        C[0][0] = a_00 * w[0] + b_00;
        C[0][1] = a_01 * w[0] + b_01;
    }

    void ComputeSubgradient(int* j_star, double* subg) {
        /* subg = sum_i a[i][j_star[i]] */
        subg[0] = (j_star[0] == 0) ? a_00 : a_01;
    }
```

Listing 5.1 Implementing the PWLData interface
After that, we need to create the launcher C++ file that will tie everything together:

```cpp
#include "data.h"
#include "solver.h"

int main(int argc, char **argv) {
    PWLData* data = new MyPWLDataImpl(-1, 1, 4, 2);

    double lambda = 1.0;
    double epsilon = 0.01;
    double bufSize = 200;
    PWLConstants* constants = new PWLConstants(lambda, epsilon, bufSize);

    PWLSolver solver(*data, *constants);

    double* w = new double[data->n]();
    solver.Solve(w);

    delete data;
    delete constants;
    delete w;
}
```

Then save, compile with g++ and run. The obtained solution is \( w = 1 \), which is indeed optimal for problem (5.1).
The approach presented in the previous section yields better performance, but lacks the user-friendliness. For the purposes of prototyping, rapid application development and for the user’s convenience we also provide the mex interface for our library. The mex interface lets the user to call our library straight from the MATLAB environment. From the user’s perspective, that approach is identical to calling any other built-in MATLAB function. Therefore, in many situations it might be more convenient than dealing with the C++ code directly.

As we have said before, the instance data of problem (1.1) is represented by $a_{ij} \in \mathbb{R}^n$ and $b_{ij} \in \mathbb{R}$, $\forall i, j$. Therefore, our mex function expects two matrices, $AA \in \mathbb{R}^{(m \cdot k) \times n}$ and $BB \in \mathbb{R}^{(m \cdot k) \times 1}$, as the input parameters, such that

$$AA(i \cdot k + j,:) = a_{ij} \text{ and } BB(i \cdot k + j,1) = b_{ij}, \forall i, j$$

Listing 5.3 shows how we can solve the problem (5.1) using the mex interface.

**Listing 5.3**  Calling the library via mex interface

```matlab
% initialize AA and BB
AA = [-1; 1];
BB = [4; 2];

% set the required constants
m = 1;
lambda = 1; epsilon = 0.01; bufSize = 200;

% call the mex function
[w, stats] = pwl(AA, BB, m, lambda, epsilon, bufSize);

% do something with solution w here ...
```

As before, the solution is $w = 1$. The underlying implementation of the mex function uses the MatlabPWLData realization of the PWLData interface that we have talked about earlier.

Listing 5.4 demonstrates the example output of calling our library via the mex interface.

**Listing 5.4**  Example output of running the algorithm via mex interface

```
Input data parameters:
    # of examples : 1
    # of dimensions : 1
Solver parameters:
    lambda : 1
    epsilon : 0.01
    preallocated planes : 200
Running the algorithm:
    001: time = 0.00003, P = 3.50000, P_t = 3.50000, (P-P_t)/P = 0.00000
Done!
Timing statistics:
    qp solver time : 0.000004
    affine time : 0.000005
    subgradient time : 0.000002
    compute w time : 0.000002
```
First, we can see the input data statistics. Then there is the information about each iteration of the algorithm, including the objective values of the master and the dual problems and the relative gap between the two. In the end, there is some information about the time it took to solve the problem.
6 Applications

In this section we will show how to use the library to implement a solver for several machine learning problems. In particular, we will demonstrate the way to formulate the problem at hand as an instance of the optimization problem (1.1) with risk function (1.2). That means the following. First, we need to define the values of the parameters $m$, $n$, $k$ and $\lambda$. Then, we have to represent the instance data via the vectors $a_{ij} \in \mathbb{R}^n$ and scalars $b_{ij} \in \mathbb{R}$, $\forall i, j$. Once the problem has the required form, we will be able to obtain a solution by letting our generic risk minimization solver do the work.

Here we present two applications: Linear Two-class SVM classifiers and Linear Multi-class SVM classifiers. We have chosen them for two reasons. First, those problems are highly popular and important in the machine learning area. Second, there already exists the implementation of the OCA algorithm (LIBOCAS [?]), which solves those two particular problems. Hence, the theory we present here will help us benchmark our algorithm and compare the results with the existing solver (Section 7).

6.1 Linear Two-class SVM classifiers

Given an example set \(\{(x_1, y_1), \ldots, (x_m, y_m)\} \in (\mathbb{R}^n \times \{-1, 1\})^m\), the goal is to find a parameter vector \(w \in \mathbb{R}^n\) of the linear classification rule \(h(x) = \text{sgn}(w, x)\) that assigns every vector \(x\) a label \(-1\) or \(+1\). The parameter vector \(w\) is obtained by solving the following unconstrained minimization problem:

\[
\begin{align*}
\mathbf{w}^* &= \arg\min_{w \in \mathbb{R}^n} \frac{1}{2} \|w\|^2 + C \sum_{i=1}^{m} \max\{0, 1 - y_i(w, x_i)\},
\end{align*}
\]

where \(C > 0\) is a regularization constant.

The formulation (6.1) can be viewed as an instance of our problem (1.1) with risk (1.2) after we apply the following data transformations:

Set

\[
\begin{align*}
\lambda &= \frac{1}{C}, \\
m &= \text{number of examples in the dataset}, \\
n &= \text{number of dimensions of the feature space}, \\
k &= 2 \ (\text{i.e., labels } -1 \text{ and } +1).
\end{align*}
\]

and for every \(i = 1, \ldots, m\), set

\[
\begin{align*}
a_{i1} &= 0, \\
a_{i2} &= -y_i x_i, \\
b_{i1} &= 0, \\
b_{i2} &= 1,
\end{align*}
\]

where \(a_{i1} \in \mathbb{R}^n, a_{i2} \in \mathbb{R}^n\) and \(b_{i1} \in \mathbb{R}, b_{i2} \in \mathbb{R}\).
6 Applications

6.2 Linear Multi-class SVM classifiers

Given an example set \( \{(x_1, y_1), \ldots, (x_m, y_m)\} \subseteq (\mathbb{R}^n \times \{1, \ldots, k\})^m \), the goal is to find a parameter vector \( w = [w_1, w_2, \ldots, w_k] \subseteq \mathbb{R}^{k \times n} \) of the linear classification rule \( h(x) = \text{argmax}_{i=1, \ldots, k} \langle w_i, x \rangle \) which assigns every example \( x \) a label from the set \( \{1, \ldots, k\} \). The parameter vector \( w \) can be obtained by solving the following unconstrained minimization problem:

\[
  w^* = \text{argmin}_{w \in \mathbb{R}^{k \times n}} \frac{1}{2} \|w\|^2 + C \sum_{i=1}^{m} \max_{y=1, \ldots, k} \left( |y \neq y_i| + \langle w_y, x_i \rangle - \langle w_{y_i}, x_i \rangle \right),
\]

(6.2)

where \( C > 0 \) is a regularization constant and \( |y \neq y_i| \) evaluates to 1 when \( y = y_i \) and it evaluates to 0 otherwise.

This optimization problem is equivalent to our problem (1.1) with risk (1.2) after we apply the following data transformations:

Set

\[
  \lambda = \frac{1}{C}, \\
  m = \text{number of examples in the dataset}, \\
  n = \text{number of dimensions of the feature space}, \\
  k = \text{number of labels in the dataset}.
\]

Then for every \( i = 1, \ldots, m \) and \( j = 1, \ldots, k \) set

\[
  a_{ij} = 0 \quad \text{and} \quad b_{ij} = 0 \quad \text{if} \quad j = y_i,
\]

and

\[
  a_{ij} = [\tilde{a}_1, \ldots, \tilde{a}_k] \quad \text{and} \quad b_{ij} = 1 \quad \text{otherwise},
\]

where \( a_{ij} \subseteq \mathbb{R}^{k \times n} \) and \( b_{ij} \subseteq \mathbb{R} \) and \( y_i \) is the label of the example \( x_i \) and

\[
  \tilde{a}_j = x_i, \\
  \tilde{a}_{y_i} = -x_i, \\
  \tilde{a}_t = 0 \quad \text{for} \quad t = 1, \ldots, k \quad \text{and} \quad t \neq j, t \neq y_i.
\]
7 Experiments

The main goal of this section is to make sure that our implementation is correct. Another goal is to verify that the convergence speed of our algorithm, despite the fact that it is fairly generic, is comparable to the speed of the specialized solvers. To do that, we will optimize several SVM problems using our library (code-named PWLSolver) and the state-of-the-art solver LIBOCAS, and compare the results. In particular, we will evaluate the outcome of the experiments based on the following criteria:

1. The number of iterations required for convergence.
2. The objective value evaluated at the optimal solution.
3. The amount of seconds required for convergence.

We have picked LIBOCAS because it implements the algorithm proposed in [17] as well, but does so for Two-class and Multi-class SVM problems only.

7.1 Data sets

The data sets used in the experiments are summarized in Table 7.1.

<table>
<thead>
<tr>
<th>Data Set</th>
<th>RNA</th>
<th>A9A</th>
<th>MNIST</th>
<th>VEHICLE</th>
<th>SHUTTLE</th>
<th>LETTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Examples</td>
<td>59,535</td>
<td>32,561</td>
<td>60,000</td>
<td>848</td>
<td>43,500</td>
<td>15,000</td>
</tr>
<tr>
<td>Dim</td>
<td>8</td>
<td>123</td>
<td>781</td>
<td>18</td>
<td>9</td>
<td>16</td>
</tr>
<tr>
<td>Classes</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>7</td>
<td>26</td>
</tr>
</tbody>
</table>

Table 7.1 Summary of the data sets used in the experiments.


The first three data sets shown in Table 7.1 have 2 classes, hence they will be used for Two-class SVM classification problems. The remaining three data sets will be used for Multi-class SVM classification. As we have mentioned before, LIBOCAS has built-in support for both types of optimization tasks. To launch our solver, we will first have to apply the data transformations described in Section 6.

We used $\epsilon = 0.01$ as a relative tolerance parameter for our stopping condition. The value of the regularization parameter $C$ varies from experiment to experiment, and its exact value can be found in the the graph descriptions in the next section.
7 Experiments

7.2 Benchmark results

The graphs illustrating the convergence process of two benchmarked algorithms are presented on Figures 7.1 to 7.6. Red line represents our implementation, and the blue line stands for LIBOCAS. The left graph of each figure shows the value of the relative gap between the value of the master problem (1.1) and the value of the reduced problem (3.1) vs. number of iterations. The graphs on the right show the dependency between the relative gap and time elapsed (in seconds).

Figure 7.1 RNA data set with $C = 0.01$.

Figure 7.2 A9A data set with $C = 1$.

Figure 7.3 MNIST data set with $C = 0.01$. 
7.2 Benchmark results

The first thing we would like to point out is that the iteration-wise convergence graphs (on the left of each figure) are almost identical for both LIBOCAS and PWLSolver implementations. In particular, this means that the solution vector and the optimal value of the objective function are the same for both solvers after each iteration of the algorithm. The number of iterations required for convergence is also the same. The only thing that differs is the convergence time. The precise results of the experiments are summarized in Table 7.2.
7 Experiments

<table>
<thead>
<tr>
<th></th>
<th>PWLSolver</th>
<th>LIBOCAS</th>
<th>PWLSolver</th>
<th>LIBOCAS</th>
<th>PWLSolver</th>
<th>LIBOCAS</th>
</tr>
</thead>
<tbody>
<tr>
<td>RNA</td>
<td>10</td>
<td>11</td>
<td>162.54</td>
<td>162.55</td>
<td>11.99</td>
<td>10.66</td>
</tr>
<tr>
<td>A9A</td>
<td>52</td>
<td>52</td>
<td>11438.33</td>
<td>11438.94</td>
<td>6.26</td>
<td>1.16</td>
</tr>
<tr>
<td>MNIST</td>
<td>3</td>
<td>3</td>
<td>5923.17</td>
<td>5923.20</td>
<td>3.35</td>
<td>0.52</td>
</tr>
<tr>
<td>VEHICLE</td>
<td>49</td>
<td>49</td>
<td>2058.66</td>
<td>2058.43</td>
<td>0.29</td>
<td>0.08</td>
</tr>
<tr>
<td>SHUTTLE</td>
<td>44</td>
<td>42</td>
<td>24071.63</td>
<td>24105.78</td>
<td>11.49</td>
<td>1.95</td>
</tr>
<tr>
<td>LETTER</td>
<td>161</td>
<td>164</td>
<td>46497.83</td>
<td>46494.93</td>
<td>260.82</td>
<td>12.33</td>
</tr>
</tbody>
</table>

Table 7.2 Exact results of the experiments.

7.3 Discussion

The equal number of iterations and the equal values of the objective function at optimum, plus the assumed correctness of LIBOCAS solver, mean that our implementation is correct as well. Therefore, we can proclaim that our effort to implement the algorithm proposed in [17] for the special case of the problem (1.1) with risk function (1.2) was successful.

On the other hand, we can see from the last two columns of Table 7.2 that the running time of PWLSolver is inferior to the running time of LIBOCAS. We have expected that kind of outcome. There are two major factors that can explain it:

1. During those experiments, we have been calling our library via the `mex` interface. That interface is completely generic, and therefore does not take into account the special structure of an optimization problem. LIBOCAS, on the other hand, exploits the special structure of the SVM problems to speed up the computations. Therefore, in practice our generic solver can never be as fast as the LIBOCAS library at solving the SVM problems.

2. The current implementation of the PWLSolver is not perfect, and there still exist parts of code that can be enhanced.

The first factor can be almost completely eliminated by writing and calling the specific implementations of the `PWLData` interface in C++, which would leverage the special structure of SVM problems to speed up the algorithm. We didn’t do that because our primary goal was to test the correctness and the performance of the generic version of our solver. Also, benchmarking via MATLAB is more convenient due to the built-in graphics toolbox and easy data manipulation.

The second factor is a known issue, and we plan to address it in the next version of the library. For more details about the future work please consult Section 8.
8 Conclusions

We have implemented the generic algorithm proposed in [17] for solving the risk minimization problems of form (1.1) with the risk function defined as in (1.2). Our main architectural goal was to make the library modular, flexible and suitable for optimizing different instances of that problem. We have achieved that by separating the constant parts of the algorithm (main computational logic) from the varying parts of the algorithm (operations involving instance data). We have demonstrated how to use our library by providing the \texttt{C++} and \texttt{mex} setup instructions, including the code snippets. After that, we have shown which data transformations are required to be able to solve the Linear Two-class SVM classification and the Linear Multi-class SVM classification problems using our library. Finally, we have compared the performance of our implementation with the performance of the state-of-the-art solver LIBOCAS. Based on the benchmark results, we have come to the conclusion that our implementation is mathematically correct.

Despite the fact that our library can already be used to solve the optimization problems in practice, there are still many parts of the algorithm that we plan to improve. In particular, the future work on the solver will include the following:

1. Implementing the more efficient algorithm for the line-search proposed in [7].
2. Adding the cutting plane management strategies.
3. Implementing the support for sparse matrices.
4. Extending the algorithm to be able to solve problems (1.1) with negative risk functions.
5. Developing the parallelized versions of the most computationally-intensive parts of the algorithm.

We also plan to do a more thorough empirical evaluation of our solver on various data sets and compare our implementation with other libraries which target the similar optimization problems.
Bibliography


[8] Thorsten Joachims. Making large-scale svm learning practical. advances in kernel methods-support vector learning, b. schölkopf and c. burges and a. smola, 1999. 4


