IPBoost – Non-Convex Boosting via Integer Programming

Marc Pfetsch [*1] **Sebastian Pokutta** [*2]

Abstract

Recently non-convex optimization approaches for solving machine learning problems have gained significant attention. In this paper we explore non-convex boosting in classification by means of integer programming and demonstrate real-world practicability of the approach while circumventing shortcomings of convex boosting approaches. We report results that are comparable to or better than the current state-of-the-art.

1. Introduction

Boosting is an important (and by now standard) technique in classification to combine several 'low accuracy' learners, so-called *base learners*, into a 'high accuracy' learner, a so-called *boosted learner*. Pioneered by the AdaBoost approach of (Freund & Schapire, 1995), in recent decades there has been extensive work on boosting procedures and analyses of their limitations. In a nutshell, boosting procedures are (typically) iterative schemes that roughly work as follows: for $t = 1, \ldots, T$ do the following:

1. Train a learner μ_t from a given class of base learners on the data distribution $\mathcal{D}_t$.
2. Evaluate performance of μ_t by computing its loss.
3. Push weight of the data distribution $\mathcal{D}_t$ towards misclassified examples leading to $\mathcal{D}_{t+1}$.

Finally, the learners are combined by some form of voting (e.g., soft or hard voting, averaging, thresholding). A close inspection of most (but not all) boosting procedures reveals that they solve an underlying convex optimization problem over a convex loss function by means of coordinate gradient descent. Boosting schemes of this type are often referred to as *convex potential boosters*. These procedures can achieve exceptional performance on many data sets if the data is correctly labeled. However, it was shown in (Long & Servedio, 2008; 2010) that any convex potential booster can be easily defeated by a very small amount of label noise (this also cannot be easily fixed by early termination). The intuitive reason for this is that convex boosting procedures might progressively zoom in on the (small percentage of) misclassified examples in an attempt to correctly label them, while simultaneously moving distribution weight away from the correctly labeled examples. As a consequence, the boosting procedure might fail and produce a boosted learner with arbitrary bad performance on unseen data.

Let $\mathcal{D} = \{(x_i, y_i) \mid i \in I\} \subseteq \mathbb{R}^d \times \{\pm 1\}$ be a set of training examples and for some logical condition C, define $\mathbb{I}[C] = 1$ if C is true and $\mathbb{I}[C] = -1$ otherwise. Typically, the true loss function of interest is of a form similar to

$$\ell(\mathcal{D}, \theta) \coloneqq \sum_{i \in I} \mathbb{I}[h_\theta(x_i) \neq y_i], \tag{1}$$

i.e., we want to minimize the number of misclassifications, where h_θ is some learner parameterized by θ; this function can be further modified to incorporate margin maximization as well as include a measure of complexity of the boosted learner to help generalization etc. It is important to observe that the loss in Equation (1) is non-convex and hard to minimize. Thus, traditionally this loss has been replaced by various convex relaxations, which are at the core of most boosting procedures. In the presence of mislabeled examples (or more generally label noise) minimizing these convex relaxations might not be a good proxy for minimizing the true loss function arising from misclassifications.

Going beyond the issue of label noise, one might ask more broadly, why not directly minimizing misclassifications (with possible regularizations) if one could? In the past, this has been out of the question due to the high complexity of minimizing the non-convex loss function. In this paper, we will demonstrate that this is *feasible and practical* with today's integer programming techniques. We propose to directly work with a loss function of the form as given in (1) (and variations) and solve the non-convex combinatorial optimization problem with state-of-the-art integer programming (IP) techniques including column generation. This

[*]Equal contribution [1]Department of Mathematics, TU Darmstadt, Germany [2]Department of Mathematics, TU Berlin and Zuse Institute Berlin, Berlin, Germany. Correspondence to: Marc Pfetsch <pfetsch@opt.tu-darmstadt.de>, Sebastian Pokutta <pokutta@zib.de>.

Proceedings of the 37^{th} International Conference on Machine Learning, Online, PMLR 119, 2020.

approach generalizes previous linear programming based approaches (and hence implicitly convex approaches) in, e.g., (Demiriz et al., 2002; Goldberg & Eckstein, 2010; 2012; Eckstein & Goldberg, 2012), while solving classification problems with the true misclassification loss. We acknowledge that (1) is theoretically very hard (in fact NP-hard as shown, e.g., in (Goldberg & Eckstein, 2010)), however, we hasten to stress that in real-world computations for *specific instances* the behavior is often much better than the theoretical asymptotic complexity. In fact, most real-world instances are actually relatively "easy" and with the availability of very strong integer programming solvers such as, e.g., the commercial solvers `CPLEX`, `Gurobi`, and `XPRESS` and the academic solver `SCIP`, these problems can be often solved rather effectively. In fact, integer programming methods have seen a huge improvement in terms of computational speed as reported in (Savickỳ et al., 2000; Bertsimas & Dunn, 2017). The latter reports that integer programming solving performance has seen a combined hardware and software speed-up of 80 billion from 1991 to 2015 (hardware: 570 000, software 1 400 000) using state-of-the-art hardware and solvers such as `CPLEX` (see (CPLEX)), `Gurobi` (see (Gurobi)), `XPRESS` (see (XPRESS)), and SCIP (see (Gamrath et al., 2020)). With this, problems that traditionally have been deemed unsolvable can be solved in reasonable short amounts of time making these methods accessible, feasible, and practical in the context of machine learning allowing to solve a (certain type of) non-convex optimization problems.

Contribution and Related Work

Our contribution can be summarized as follows:

IP-based boosting. We propose an integer programming based boosting procedure. The resulting procedure utilizes column generation to solve the initial learning problem and is inherently robust to labeling noise, since we solve the problem for the (true) non-convex loss function. In particular, our procedure is robust to the instances from (Long & Servedio, 2008; 2010) that defeat other convex potential boosters.

Linear Programming (LP) based boosting procedures have been already explored with *LPBoost* (Demiriz et al., 2002), which also relies on column generation to price the learners. Subsequent work in (Leskovec & Shawe-Taylor, 2003) considered LP-based boosting for uneven datasets. We also perform column generation, however, in an IP framework (see (Desrosiers & Lübbecke, 2005) for an introduction) rather than a purely LP-based approach, which significantly complicates things. In order to control complexity, overfitting, and generalization of the model typically some sparsity is enforced. Previous approaches in the context of LP-based boosting have promoted sparsity by means of cutting planes, see, e.g., (Goldberg & Eckstein, 2010; 2012; Eckstein & Goldberg, 2012). Sparsification can be handled in our approach by solving a delayed integer program using additional cutting planes.

An interesting alternative use of boosting in the context of training average learners against rare examples has been explored in (Shalev-Shwartz & Wexler, 2016); here the 'boosting' of the data distribution is performed *while* a more complex learner is trained. In (Freund et al., 2015) boosting in the context of linear regression has been shown to reduce to a certain form of subgradient descent over an appropriate loss function. For a general overview of boosting methods we refer the interested reader to (Schapire, 2003). Non-convex approaches to machine learning problems gained recent attention and (mixed) integer programming, in particular, has been used successfully to incorporate combinatorial structure in classification, see, e.g., (Bertsimas & Shioda, 2007; Chang et al., 2012; Bertsimas & King, 2015; Bertsimas et al., 2016), as well as, (Günlük et al., 2016; Bertsimas & Dunn, 2017; Dash et al., 2018; Günlük et al., 2018; Verwer & Zhang, 2019); note that (Dash et al., 2018) also uses a column generation approach. Moreover, neural network verification via integer programming has been treated in (Tjeng et al., 2017; Fischetti & Jo, 2018). See also the references contained in all of these papers.

Computational results. We present computational results demonstrating that IP-based boosting can avoid the bad examples of (Long & Servedio, 2008): by far better solutions can be obtained via LP/IP-based boosting for these instances. We also show that IP-based boosting can be competitive for real-world instances from the LIBSVM data set. In fact, we obtain nearly optimal solutions in reasonable time for the true non-convex cost function. Good solutions can be obtained if the process is stopped early. While it cannot match the raw speed of convex boosters, the obtained results are (often) much better. Moreover, the resulting solutions are often sparse.

2. IPBoost: Boosting via Integer Programming

We will now introduce the basic formulation of our boosting problem, which is an integer programming formulation based on the standard LPBoost model from (Demiriz et al., 2002). While we confine the exposition to the binary classification case only, for the sake of clarity and due to space limitations, we stress that our approach can be extended to the multi-class case using standard methods. In subsequent sections, we will refine the model to include additional model parameters etc.

Let $(x_1, y_1), \ldots, (x_N, y_N)$ be the training set with points $x_i \in \mathbb{R}^d$ and two-class labels $y_i \in \{\pm 1\}$. Moreover, let

$\Omega \coloneqq \{h_1, \ldots, h_L : \mathbb{R}^d \to \{\pm 1\}\}$ be a class of base learners and let a margin $\rho \geq 0$ be given. Our basic boosting model is captured by the following integer programming problem:

$$\begin{aligned} \min \ & \sum_{i=1}^{N} z_i && (2) \\ & \sum_{j=1}^{L} \eta_{ij}\, \lambda_j + (1+\rho) z_i \geq \rho \quad \forall\, i \in [N], \\ & \sum_{j=1}^{L} \lambda_j = 1,\ \lambda \geq 0, \\ & z \in \{0,1\}^N, \end{aligned}$$

where the *error function* η can take various forms depending on how we want to treat the output of base learners. For learner h_j and training example x_i we consider the following choices:

(i) ± 1 classification from learners:
$\eta_{ij} \coloneqq 2\,\mathbb{I}[h_j(x_i) = y_i] - 1 = y_i \cdot h_j(x_i)$;

(ii) class probabilities of learners:
$\eta_{ij} \coloneqq 2\,\mathbb{P}[h_j(x_i) = y_i] - 1$;

(iii) `SAMME.R` error function for learners:
$\eta_{ij} \coloneqq \frac{1}{2} y_i \log\left(\frac{\mathbb{P}[h_j(x_i)=1]}{\mathbb{P}[h_j(x_i)=-1]}\right)$.

In the first case we perform a hard minimization of the classification error, in the second case we perform a soft minimization of the classification error, and in the last one we minimize the `SAMME.R` error function as used in the (multi-class) AdaBoost variant in (Zhu et al., 2009). The `SAMME.R` error function allows a very confident learner to overrule a larger number of less confident learners predicting the opposite class.

The z_i variable in the model above indicates whether example $i \in [N] \coloneqq \{1, \ldots N\}$ satisfies the classification requirement: $z_i = 0$ if example i is correctly labeled by the boosted learner $\sum_j h_j \lambda_j$ with margin at least ρ with respect to the utilized error function η; in an optimal solution, if a variable is 1 this implies misclassification, otherwise by minimizing you could have set it to zero. The λ_j with $j \in [L]$ form a distribution over the family of base learners. The only non-trivial family of inequalities in (2) ranges over examples $i \in [N]$ and enforces that the combined learner $\sum_{j \in [L]} h_j \lambda_j$ classifies example $i \in N$ correctly with margin at least ρ (we assume throughout that $\rho \leq 1$) or $z_i = 1$, i.e., the example is disregarded and potentially misclassified. By minimizing $\sum_{i \in N} z_i$, the program computes the best combination of base learners maximizing the number of examples that are correctly classified with margin at least ρ. The margin parameter ρ helps generalization as it prevents base learners to be used to explain low-margin noise.

Before we continue with the integer programming based boosting algorithm we would like to remark the following about the solution structure of optimal solutions with respect to the chosen margin:

Lemma 1 (Structure of high-margin solutions) *Let (λ, z) be an optimal solution to the integer program (2) for a given margin ρ using error function (i). Further let $I \coloneqq \{i \in [N] \mid z_i = 0\}$ and $J \coloneqq \{j \in [L] \mid \lambda_j > 0\}$. If the optimal solution is non-trivial, i.e., $I \neq \emptyset$, then the following holds:*

1. *If $\rho = 1$, then there exists an optimal solution with margin 1 using only a single base learner h_j for some $j \in J$.*
2. *If there exists $\bar{\jmath} \in J$ with $\lambda_{\bar{\jmath}} > \frac{1-\rho}{2}$, then $h_{\bar{\jmath}}$ by itself is already an optimal solution with margin 1.*
3. *If $|J| < \frac{2}{1-\rho}$, then there exists $\bar{\jmath} \in J$ with $h_{\bar{\jmath}}$ by itself being already an optimal solution with margin 1. In particular for $\rho > 0$, the statement is non-trivial.*

Proof. For the first case observe that

$$\sum_{j \in J} \eta_{ij} \lambda_j \geq 1,$$

holds for all $i \in I$. As $\sum_{j \in J} \lambda_j = 1$ and $\lambda_j > 0$ for all $j \in J$, we have that $\eta_{ij} = 1$ for all $i \in I$, $j \in J$. Therefore the predictions of all learners h_j with $j \in J$ for examples $i \in I$ are identical and we can simply choose any such learner h_j with $j \in J$ arbitrarily and set $\lambda_j = 1$.

For the second case observe as before that $\sum_{j \in J} \eta_{ij} \lambda_j \geq \rho$ holds for all $i \in I$. We claim that $\eta_{i\bar{\jmath}} = 1$ for all $i \in I$. For contradiction suppose not, i.e., there exists $\bar{\imath} \in I$ with $\eta_{\bar{\imath}\bar{\jmath}} = -1$. Then

$$\sum_{j \in J} \eta_{\bar{\imath} j} \lambda_j < \underbrace{\left(\sum_{j \in J \setminus \{\bar{\jmath}\}} \lambda_j\right)}_{< 1 - \frac{1-\rho}{2}} - \frac{1-\rho}{2} < \rho,$$

using $\eta_{ij} \leq 1$, $\sum_{j \in J} \lambda_j = 1$, and $\lambda_j > 0$ for all $j \in J$. This contradicts $\sum_{j \in J} \eta_{ij} \lambda_j \geq \rho$ and therefore $\eta_{i\bar{\jmath}} = 1$ for all $i \in I$. Thus $h_{\bar{\jmath}}$ by itself is already an optimal solution satisfying even the (potentially) higher margin of $1 \geq \rho$ on examples $i \in I$.

Finally, for the last case observe that if $|J| < \frac{2}{1-\rho}$, then together with $\sum_{j \in J} \lambda_j = 1$, and $\lambda_j > 0$ for all $j \in J$, it follows that there exists $\bar{\jmath} \in J$ with $\lambda_{\bar{\jmath}} > \frac{1-\rho}{2}$. Otherwise $\sum_{j \in J} \lambda_j \leq |J| \frac{1-\rho}{2} < 1$; a contradiction. We can now apply the second case. □

Similar observations hold for error functions *(ii)* and *(iii)* with the obvious modifications to include the actual value of η_{ij} not just its sign.

Our proposed solution process consists of two parts. We first solve the integer program in (2) using column generation. Once this step is completed, the solution can be sparsified (if necessary) by means of the model presented in Section 2.2, where we trade-off classification performance with model complexity.

2.1. Solution Process using Column Generation

The reader will have realized that (4) is not practical, since we typically have a very large if not infinite class of base learners Ω; for convenience we assume here that Ω is finite but potentially very large. This has been already observed before and dealt with effectively via column generation in (Demiriz et al., 2002; Goldberg & Eckstein, 2010; 2012; Eckstein & Goldberg, 2012). We will follow a similar strategy here, however, we generate columns *within* a branch-and-bound framework leading effectively to a *branch-and-bound-and-price* algorithm that we are using; this is significantly more involved compared to column generation in linear programming. We detail this approach in the following.

The goal of column generation is to provide an efficient way to solve the linear programming (LP) relaxation of (2), i.e., the z_i variables are relaxed and allowed to assume fractional values. Moreover, one uses a subset of the columns, i.e., base learners, $\mathcal{L} \subseteq [L]$. This yields the so-call *restricted master (primal) problem*

$$\begin{aligned} \min \ & \sum_{i=1}^{N} z_i \qquad (3) \\ & \sum_{j\in\mathcal{L}} \eta_{ij}\,\lambda_j + (1+\rho) z_i \geq \rho \quad \forall\, i \in [N], \\ & \sum_{j\in\mathcal{L}} \lambda_j = 1,\ \lambda \geq 0,\ z \in [0,1]^N. \end{aligned}$$

Its *restricted dual problem* is

$$\begin{aligned} \max\ & \rho \sum_{i=1}^{N} w_i + v - \sum_{i=1}^{N} u_i \qquad (4) \\ & \sum_{i=1}^{N} \eta_{ij}\, w_i + v \leq 0 \quad \forall\, j \in \mathcal{L}, \\ & (1+\rho) w_i - u_i \leq 1 \quad \forall\, i \in [N], \\ & w \geq 0,\ u \geq 0,\ v \text{ free}. \end{aligned}$$

Consider a solution $(w^*, v^*, u^*) \in \mathbb{R}^N \times \mathbb{R} \times \mathbb{R}^N$ of (4). The so-called *pricing problem* is to decide whether this solution is actually optimal or whether we can add further constraints, i.e., columns in the primal problem. For this, we need to check whether (w^*, v^*, u^*) is feasible for the complete set of constraints in (4). In the following, we will assume that the variables z_i are always present in the primal and therefore that the corresponding inequalities $(1+\rho)w_i - u_i \leq 1$ are satisfied for each $i \in [N]$. Thus, the main task of the pricing problem is to decide whether there exists $j \in [L] \setminus \mathcal{L}$ such that

$$\sum_{i=1}^{N} \eta_{ij}\, w_i^* + v^* > 0. \qquad (5)$$

If such an j exists, then it is added to $\mathcal{L}$, i.e., to (3), and the process is iterated. Otherwise, both (3) and (4) have been solved to optimality.

The pricing problem (5) can now be rephrased as follows: Does there exist a base learner $h_j \in \Omega$ such that (5) holds? For this, the w_i^* can be seen as weights over the points x_i, $i \in [N]$, and we have to classify the points according to these weights. For most base learners, this task just corresponds to an ordinary classification or regression step, depending on the form chosen for η_{ij}. Note, however, that in practice (5) is not solved to optimality, but is rather solved heuristically. If we find a base learner h_j that satisfies (5), we continue, otherwise we stop.

The process just described allows to solve the relaxation of (2). The optimal misclassification values are determined by a branch-and-price process that branches on the variables z_i and solves the intermediate LPs using column generation. Note that the z_i are always present in the LP, which means that no problem-specific branching rule is needed, see, e.g., (Barnhart et al., 1998) for a discussion. In total, this yields Algorithm 1.

The output of the algorithm is a set of base learners $\mathcal{L}^*$ and corresponding weights λ_j^*. A classifier can be obtained by *voting*, i.e., a given point x is classified by each of the base learners resulting in ξ_j, for which we again can use the three options *(i)*–*(iii)* above. We then take the weighted combination and obtain the predicted label as $\operatorname{sgn}(\sum_{j\in\mathcal{L}^*} \xi_j \lambda_j^*)$.

In the implementation, we use the following important components. First, we use the framework SCIP that automatically applies primal heuristics, see, e.g., (Berthold, 2014) for an overview. These heuristics usually take the current solution of the relaxation and try to build a feasible solution for (2). In the current application, the most important heuristics are rounding heuristics, i.e., the z_i variables are rounded to 0 or 1, but large-scale neighborhood heuristics sometimes provide very good solutions as well. Nevertheless, we disable diving heuristics, since these often needed a long time, but never produced a feasible solution. In total, this often generates many feasible solutions along the way.

Another trick that we apply is the so-called *stall limit*. The

Algorithm 1 IPBoost

Input: Examples $\mathcal{D} = \{(x_i, y_i) \mid i \in I\} \subseteq \mathbb{R}^d \times \{\pm 1\}$, class of base learners Ω, margin ρ
Output: Boosted learner $\sum_{j \in \mathcal{L}^*} h_j \lambda_j^*$ with base learners h_j and weights λ_j^*

$\mathcal{T} \leftarrow \{([0,1]^N, \varnothing)\}$ // set of local bounds and learners for open subproblems
$U \leftarrow \infty$, $\mathcal{L}^* \leftarrow \varnothing$ // upper bound on optimal objective
while $\mathcal{T} \neq \varnothing$ **do**
  Choose and remove $(B, \mathcal{L})$ from $\mathcal{T}$.
  repeat
    Solve (3) using the local bounds on z in B with optimal dual solution (w^*, v^*, u^*).
    Find learner $h_j \in \Omega$ satisfying (5). // solve pricing problem
  until h_j is not found
  Let $(\tilde{\lambda}, \tilde{z})$ be the final solution of (3) with base learners $\tilde{\mathcal{L}} = \{j \mid \tilde{\lambda}_j > 0\}$.
  if $\sum_{i=1}^N \tilde{z}_i < U$ **then**
    if $\tilde{z} \in \{0,1\}^N$ **then**
      $U \leftarrow \sum_{i=1}^N \tilde{z}_i$, $\mathcal{L}^* \leftarrow \tilde{\mathcal{L}}$, $\lambda^* \leftarrow \tilde{\lambda}$ // update best sol.
    else
      Choose $i \in [N]$ with $\tilde{z}_i \notin \mathbb{Z}$.
      Set $B_0 \leftarrow B \cap \{z_i \leq 0\}$, $B_1 \leftarrow B \cap \{z_i \geq 1\}$.
      Add $(B_0, \tilde{\mathcal{L}})$, $(B_1, \tilde{\mathcal{L}})$ to $\mathcal{T}$. // create new branching nodes
    end if
  end if
end while
Optionally sparsify final solution $\mathcal{L}^*$.

solver automatically stops if the best primal solution could not be (strictly) improved during the last K nodes processed in the branch-and-bound tree (we use $K = 5000$).

Furthermore, preliminary experiments have shown that the intermediate linear programs that have to be solved in each iteration become increasingly hard to solve by the simplex algorithm for a large number of training points. We could apply bagging (Breiman, 1996), but obtained good results with just subsampling 30 000 points if their number N is larger than this threshold.

Furthermore, we perform the following post-processing. For the best solution that is available at the end of the branch-and-bound algorithm, we fix the integer variables to the values in this solution. Then we maximize the margin over the learner variables that were used in the solution, which is just a linear program. In most cases, the margin can be slightly improved in this way, hoping to get improved generalization.

2.2. Sparsification

One of the challenges in boosting is to balance model accuracy vs. model generalization, i.e., to prevent overfitting. Apart from pure generalization considerations, a sparse model often lends itself more easily to interpretation, which might be important in certain applications.

There are essentially two techniques that are commonly used in this context. The first one is early stopping, i.e., we only perform a fixed number of boosting iterations, which would correspond to only generating a fixed number of columns. The second common approach is to regularize the problem by adding a complexity term for the learners in the objective function, so that we minimize $\sum_{i=1}^N z_i + \sum_{j=1}^L \alpha_j y_j$. Then we can pick α_j as a function of the complexity of the learner h_j. For example, in (Cortes et al., 2014) boosting across classes of more complex learners has been considered and the α_j are chosen to be proportional to the Rademacher complexity of the learners (many other measures might be equally justified).

In our context, it seems natural to consider the following integer program for sparsification:

$$\begin{aligned} \min \ & \sum_{i=1}^N z_i + \sum_{j=1}^L \alpha_j y_j \\ & \sum_{j=1}^L \eta_{ij} \lambda_j + (1+\rho) z_i \geq \rho \quad \forall\, i \in [N], \\ & \sum_{j=1}^L \lambda_j = 1,\ 0 \leq \lambda_j \leq y_j \quad \forall\, j \in [L], \\ & z \in \{0,1\}^N,\ y \in \{0,1\}^L, \end{aligned}$$

with η_{ij} as before. The structure of this sparsification problem that involves additional binary variables y cannot be easily represented within the column generation setup used to solve model (2), because the upper bounds on λ_j implied by y_j would need to be represented in dual problem, giving rise to exponentially many variables in the dual. In principle, one could handle a cardinality constraint on the y_j variables using a problem specific branching rule; this more involved algorithm is however beyond the scope of this paper. In consequence, one can solve the sparsification problem separately for the columns that have been selected in phase 1 once this phase is completed. This is similar to (Goldberg & Eckstein, 2010), but directly aims to solve the MIP rather than a relaxation. Moreover, one can apply so-called IIS-cuts, following (Pfetsch, 2008). Using the Farkas lemma, the idea is to identify subsets $I \subseteq [N]$ such that the system

$$\sum_{j=1}^L \eta_{ij} \lambda_j \geq \rho,\ i \in I, \quad \sum_{j=1}^L \lambda_j = 1,\ \lambda \geq 0,$$

is infeasible. In this case the cut

$$\sum_{i \in I} z_i \geq 1$$

is valid. Such sets I can be found by searching for vertices of the corresponding alternative polyhedron. If this is done iteratively (see (Pfetsch, 2008)), many such cuts can be found that help to strengthen the LP relaxation. These cuts dominate the ones in (Goldberg & Eckstein, 2010), but one needs to solve an LP for each vertex/cut.

3. Computational Results

To evaluate the performance of IPBoost, we ran a wide range of tests on various classification tasks. Due to space limitations, we will only be able to report aggregate results here; additional more extensive results can be found in the Supplementary Material. The code is available through the web pages of the authors.

Computational Setup. All tests were run on a Linux cluster with Intel Xeon quad core CPUs with 3.50GHz, 10 MB cache, and 32 GB of main memory. All runs were performed with a single process per node; we stress, in particular, that we run all tests as *single thread / single core* setup, i.e., each test uses only a single node in single thread mode. We used a prerelease version of SCIP 7.0.0 with SoPlex 5.0.0 as LP-solver (Gamrath et al., 2020); note that this combination is completely available in source code and free for academic use at `www.scipopt.org`. The main part of the code was implemented in C, calling the python framework `scikit-learn` (Pedregosa et al., 2011) at several places. We use the decision tree implementation of `scikit-learn` with a maximal depth of 1, i.e., a decision stump, as base learners for all boosters. We benchmarked IPBoost against our own implementation of LPBoost (Demiriz et al., 2002) as well as the AdaBoost implementation in version 0.21.3 of `scikit-learn` using 100 iterations; note that we always report the number of pairwise distinct base learners for AdaBoost. We performed 10 runs for each instance with varying random seeds (which for instance affects randomly chosen test sets) and we report average accuracy and standard deviations. Note that we use a time limit of one hour for each run of IPBoost. The reported solution is the best solution available at that time.

Results on Constructed Hard Instances. We start our discussion of computational results by reporting on experiments with the hard instances of (Long & Servedio, 2008). These examples are tailored to using the ± 1 classification from learners (option *(i)* in Section 2). Thus, we use this function for prediction and voting for every algorithm. The performance of IPBoost, LPBoost and AdaBoost (using 100 iterations) is presented in Table 1. Here, N is the number of points and γ refers to the noise level. Note that we randomly split off 20 % of the points for the test set, and recall that we report the averages of 10 runs.

On every instance class, IPBoost clearly outperforms LPBoost. AdaBoost performs much less well, as expected; it also uses significantly more base learners. Note, however, that the `scikit-learn` implementation of AdaBoost produces much better results than the one in (Long & Servedio, 2008) (an accuracy of about 53 % as opposed to 33 %). As noted above, the instances are constructed for a ± 1 classification function. If we change to `SAMME.R`, AdaBoost performs much better: slightly worse that IPBoost, but better than LPBoost.

LIBSVM Instances. We use classification instances from the LIBSVM data sets available at `https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/`. We selected the 40 smallest instances. If available, we choose the scaled version over the unscaled version. Note that 25 instances of those 40 instances come with a corresponding test set. Since the test sets for the instances `a1a`–`a9a` looked suspicious (often more features and points than in the train set and sometimes only features in one class), we decided to remove the test sets for these nine instances. This leaves 16 instances with test set. For the other 24, we randomly split off 20% of the points as a test set; we provide statistics for the individual instances in the Supplementary Material.

Results for LIBSVM. An important choice for the algorithm is how the error matrix η is set up, i.e., which of the three options *(i)*–*(iii)* presented in Section 2 is used. In preliminary computations, we compared all three possibilities. It turned out that the best option is to use the class probabilities *(ii)* for η both for Model (2) and when using the base learners in a voting scheme, which we report here.

Another crucial choice in our approach is the margin bound ρ. We ran our code with different values – the aggregated results are presented in Table 2; the detailed results are given in the Supplementary Material. We report accuracies on the test set and train set, respectively. In each case, we report the averages of the accuracies over 10 runs with a different random seed and their standard deviations. The accuracies of IPBoost are compared to LPBoost and AdaBoost. We also report the number L of learners in the detailed results. Note that the behavior of AdaBoost is independent of ρ, i.e., the accuracies are the same over the different values of ρ in Table 2.

The results show that IPBoost outperforms both LPBoost and AdaBoost. IPBoost clearly outperforms LPBoost, although there are instances where LPBoost generates slightly better results, both for the train and the test accuracies. Interestingly, the accuracies of IPBoost (and LPBoost) increase with respect to AdaBoost, when considering the test set instead of the training set: less overfitting and better generalization. For the considered instances the best value for the

Table 1. Averages of the *test* accuracies for *hard instances*. The table shows the accuracies and standard deviations as well as the number of learners L for three algorithms using $\rho = 0.05$ for 10 different seeds; best solutions are marked with *; using ± 1 *values for prediction and voting*.

		IPBoost			LPBoost			AdaBoost	
N	γ		score	L		score	L	score	L
2000	0.1	*	69.05 ± 2.54	4.8		66.22 ± 1.73	2.0	58.58 ±2.77	20.9
4000	0.1	*	68.61 ± 1.50	4.6		65.23 ± 1.95	2.0	55.45 ±2.99	20.9
8000	0.1	*	67.26 ± 1.62	3.6		64.58 ± 1.05	2.0	53.24 ±1.68	20.9
16000	0.1	*	67.50 ± 1.48	3.3		64.73 ± 0.80	2.0	51.85 ±0.80	21.0
32000	0.1	*	67.36 ± 1.55	2.6		65.18 ± 0.55	2.0	51.22 ±0.73	20.9
64000	0.1	*	66.65 ± 1.04	2.5		65.17 ± 0.35	2.0	50.48 ±0.49	20.9
2000	0.075	*	71.30 ± 2.06	4.6		66.55 ± 1.89	2.0	57.95 ±2.83	21.1
4000	0.075	*	70.20 ± 1.69	4.1		66.54 ± 1.58	2.0	55.27 ±2.77	21.0
8000	0.075	*	68.41 ± 1.73	3.8		65.38 ± 1.05	2.0	53.14 ±1.51	21.0
16000	0.075	*	68.10 ± 2.18	2.9		65.63 ± 0.81	2.0	51.73 ±0.67	21.0
32000	0.075	*	68.06 ± 1.47	2.6		66.17 ± 0.62	2.0	51.12 ±0.61	20.9
64000	0.075	*	67.92 ± 1.05	2.4		66.12 ± 0.33	2.0	50.35 ±0.47	21.0
2000	0.05	*	72.20 ± 1.92	5.1		67.05 ± 1.71	2.0	57.50 ±2.51	21.0
4000	0.05	*	71.74 ± 1.59	4.9		67.27 ± 1.69	2.0	54.75 ±2.47	20.9
8000	0.05	*	70.09 ± 1.96	3.4		66.19 ± 1.22	2.0	53.01 ±1.40	21.0
16000	0.05	*	70.05 ± 1.57	3.3		66.82 ± 0.81	2.0	51.75 ±0.85	21.0
32000	0.05	*	69.25 ± 1.86	2.4		67.30 ± 0.54	2.0	51.15 ±0.65	21.0
64000	0.05	*	68.83 ± 1.44	2.3		67.06 ± 0.37	2.0	50.35 ±0.54	21.0
averages:		18	69.03 ± 1.68	3.5	0	66.07 ± 1.06	2.0	0 53.27 ±1.49	21.0

margin ρ was 0.05 for LPBoost and IPBoost; AdaBoost has no margin parameter.

Depending on the size of the instances, typical running times of IPBoost range from a few seconds up to one hour. We provide details of the running times in the Supplementary Material. The average run time of IPBoost for $\rho = 0.05$ is 1367.78 seconds, while LPBoost uses 164.35 seconds, and AdaBoost 3.59 seconds. The main bottleneck arises from the solution of large LP relaxations in each iteration of the algorithm. Note that we apply the simplex algorithm in order to benefit from hot start after changing the problem by adding columns or changing bounds. Nevertheless, larger LPs turned out to be very hard to solve. One explanation for this is that the matrix is very dense.

Feasible solutions of high quality are often found after a few seconds via primal heuristics. The solution that is actually used for constructing the boosted learner is often found long before the solution process finished, i.e., the algorithm continues to search for better solutions without further progress. Note that in most cases, the algorithm is stopped, because no further improving solution was found, i.e., the stall limit is applied (see Section 2.1). We have experimented with larger limits ($K > 5000$), but the quality of the solutions only improved very slightly. This suggests that the solutions we found are optimal or close to optimal for (2).

Also interesting is the number of base learners in the best solutions of IPBoost. The results show that this is around 12 on average for $\rho = 0.05$; for $\rho = 0.01$ it is around 18. Thus, the optimal solutions are inherently sparse. Therefore, for these settings and instances, the sparsification procedure described in Section 2.2 will likely not be successful. However, it seems likely that for instance sets requiring different margins the situation is different.

We have also experimented with different ways to handle ρ. Following (Demiriz et al., 2002), one can set up a model in which the margin ρ is a variable to be optimized in addition to the number of misclassifications. In this model, it is crucial to find the right balance between the different parts of the objective. For instance, on can run some algorithm (AdaBoost) to estimate the number of misclassifications and then adjust the weight factor accordingly. In preliminary experiments, this option was inferior to the approach described in this paper; we used an identical approach for LPBoost for consistency.

Generalization Performance. We found that the boosted learners computed via IPBoost generalize rather well. Figure 1 gives a representative example for generalization: here we plot the train and test accuracy of the solutions encountered by IPBoost within a run, while solving the boosting problem for various margins.

We observe the following almost monotonous behavior: the smaller ρ, the more base learners are used and the better the obtained training accuracy. This is of course expected, since smaller margins allow more freedom to combine base learners. However, this behavior does not directly translate to better testing accuracies, which indicates overfitting.

Table 2. Aggregated results for LIBSVM: Average test/train accuracies and standard deviations (STD) for three algorithms over 10 different seeds, using class probabilities for prediction and voting; we considered 40 instances as outlined in Section 3. Column "# best" represents the number of instances on which the corresponding algorithm performed best (ties possible). Column "ER" gives the error rate, i.e., $1/(1-a)$ for the average accuracy a.

		IPBoost				LPBoost				AdaBoost			
type	ρ	# best	accuracy	STD	ER	# best	accuracy	STD	ER	# best	accuracy	STD	ER
test	0.1	24	80.70	4.08	5.18	7	80.16	3.93	5.04	9	79.79	3.82	4.95
test	0.075	25	80.77	3.89	5.20	6	80.21	3.89	5.05	9	79.79	3.82	4.95
test	0.05	26	80.78	4.07	5.20	7	80.31	3.73	5.08	8	79.79	3.82	4.95
test	0.025	25	80.63	3.94	5.16	7	80.21	3.91	5.05	8	79.79	3.82	4.95
test	0.01	26	80.59	3.91	5.15	7	79.80	3.67	4.95	7	79.79	3.82	4.95
train	0.1	25	83.52	2.51	6.07	1	82.29	2.61	5.65	15	84.36	1.99	6.40
train	0.075	24	83.94	2.38	6.23	1	82.52	2.54	5.72	15	84.36	1.99	6.40
train	0.05	26	84.34	2.43	6.38	1	82.90	2.40	5.85	14	84.36	1.99	6.40
train	0.025	29	84.97	2.44	6.65	1	83.39	2.43	6.02	10	84.36	1.99	6.40
train	0.01	31	85.69	2.48	6.99	3	84.20	2.26	6.33	6	84.36	1.99	6.40

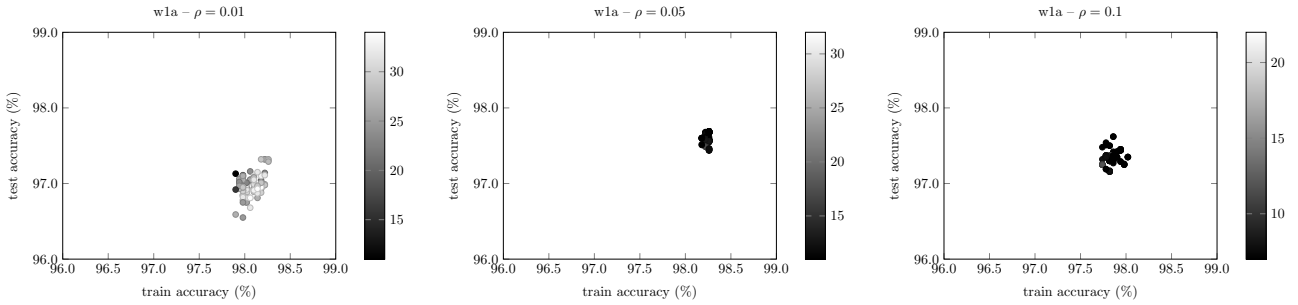

Figure 1. Train vs. test performance for different margins ρ on instance `w1a`. Each point represents a solution encountered by IPBoost while solving the boosting problem. Grayscale values indicate the number of base learners used in boosted learners; see the legend.

Note that IPBoost obtains better average test accuracies than AdaBoost for every ρ, but this is not always the case for the train accuracies. This again demonstrates the good generalization properties of IPBoost.

We would also like to point out that the results in Figures 1, 2, and 3 give an indication that the often cited belief that "solving (close) to optimality reduces generalization" is not true in general. In fact, minimizing the right loss function close to optimality can actually *help* generalization.

4. Concluding Remarks

In this paper, we have first reproduced the observation that boosting based on column generation, i.e., LP- and IP-boosting, avoids the bad performance on the well-known hard classes from the literature. More importantly, we have shown that IP-boosting improves upon LP-boosting and AdaBoost on the LIBSVM instances on which a consistent improvement even by a few percent is not easy. The price to pay is that the running time with the current implementation is much longer. Nevertheless, the results are promising, so it can make sense to tune the performance, e.g., by solving the intermediate LPs only approximately and deriving tailored heuristics that generate very good primal solutions, see (Borndörfer et al., 2008) and (Borndörfer et al., 2013), respectively, for examples for column generation in public transport optimization. Another direction is that the instances have low treewidth, which make the LP-relaxations amendable to decomposition approaches.

Moreover, our method has a parameter that needs to be tuned, namely the margin bound ρ. It shares this property with LP-boosting, where one either needs to set ρ or a corresponding objective weight. AdaBoost, however, depends on the number of iterations which also has to be adjusted to the instance set. We plan to investigate methods based on the approach in the current paper that avoid the dependence on a parameter.

In conclusion, our approach is suited very well to an offline setting in which training may take time and where even a small improvement is beneficial or when convex boosters behave very badly. Moreover, it can serve as a tool to investigate the general performance of such methods.

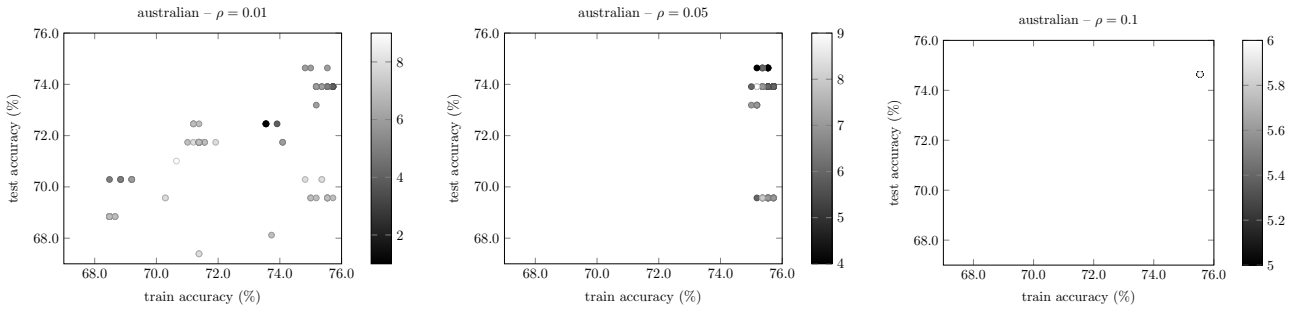

Figure 2. Train vs. test performance for different margins ρ on instance `australian`. Each point represents a solution encountered by IPBoost while solving the boosting problem. Grayscale values indicate the number of base learners used in boosted learners; see the legend.

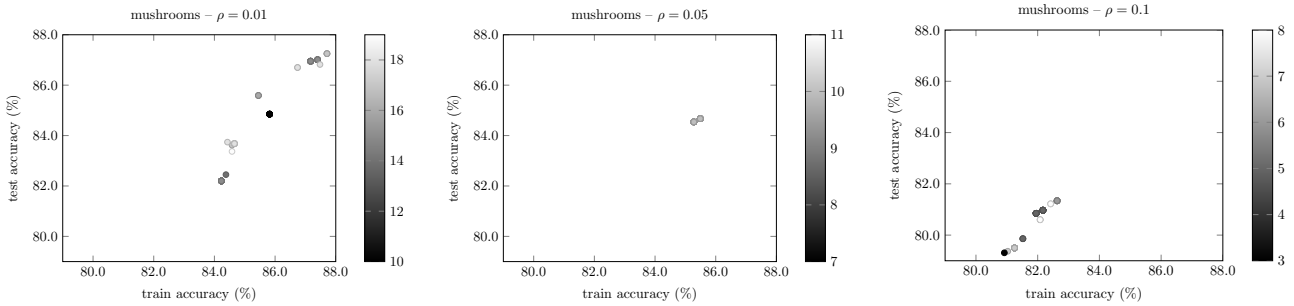

Figure 3. Train vs. test performance for different margins ρ on instance `mushrooms`. Each point represents a solution encountered by IPBoost while solving the boosting problem. Grayscale values indicate the number of base learners used in boosted learners; see the legend.

5. Acknowledgements

We thank the reviewers for their helpful comments. We also would like to thank Steve Aoki for providing inspiration in the late stages of this work.

References

Barnhart, C., Johnson, E. L., Nemhauser, G. L., Savelsbergh, M. W. P., and Vance, P. H. Branch-and-price: Column generation for solving huge integer programs. *Oper. Res.*, 46(3):316–329, 1998.

Berthold, T. *Heuristic algorithms in global MINLP solvers*. PhD thesis, TU Berlin, 2014.

Bertsimas, D. and Dunn, J. Optimal classification trees. *Machine Learning*, pp. 1–44, 2017.

Bertsimas, D. and King, A. OR forum—an algorithmic approach to linear regression. *Operations Research*, 64(1):2–16, 2015.

Bertsimas, D. and Shioda, R. Classification and regression via integer optimization. *Operations Research*, 55(2):252–271, 2007.

Bertsimas, D., King, A., Mazumder, R., et al. Best subset selection via a modern optimization lens. *The Annals of Statistics*, 44(2): 813–852, 2016.

Borndörfer, R., Löbel, A., and Weider, S. A bundle method for integrated multi-depot vehicle and duty scheduling in public transit. In Hickman, M., Mirchandani, P., and Voß, S. (eds.), *Computer-aided Systems in Public Transport*, volume 600, pp. 3–24, 2008.

Borndörfer, R., Löbel, A., Reuther, M., Schlechte, T., and Weider, S. Rapid branching. *Public Transport*, 5(1):3–23, 2013.

Breiman, L. Bagging predictors. *Machine Learning*, 24:123–140, 1996. doi: 10.1007/BF00058655.

Chang, A., Bertsimas, D., and Rudin, C. An integer optimization approach to associative classification. In *Advances in Neural Information Processing Systems*, pp. 269–277, 2012.

Cortes, C., Mohri, M., and Syed, U. Deep boosting. In *31st International Conference on Machine Learning, ICML 2014*. International Machine Learning Society (IMLS), 2014.

CPLEX. IBM ILOG CPLEX Optimizer. https://www.ibm.com/analytics/cplex-optimizer, 2020.

Dash, S., Günlük, O., and Wei, D. Boolean decision rules via column generation. In *Advances in Neural Information Processing Systems*, pp. 4655–4665, 2018.

Demiriz, A., Bennett, K. P., and Shawe-Taylor, J. Linear programming boosting via column generation. *Machine Learning*, 46 (1-3):225–254, 2002.

Desrosiers, J. and Lübbecke, M. E. A primer in column generation. In *Column generation*, pp. 1–32. Springer, 2005.

Eckstein, J. and Goldberg, N. An improved branch-and-bound method for maximum monomial agreement. *INFORMS Journal on Computing*, 24(2):328–341, 2012.

Fischetti, M. and Jo, J. Deep neural networks and mixed integer linear optimization. *Constraints*, 23(3):296–309, 2018.

Freund, R. M., Grigas, P., and Mazumder, R. A new perspective on boosting in linear regression via subgradient optimization and relatives. *Preprint arXiv:1505.04243*, 2015.

Freund, Y. and Schapire, R. E. A desicion-theoretic generalization of on-line learning and an application to boosting. In *European Conference on Computational Learning Theory*, pp. 23–37. Springer, 1995.

Gamrath, G., Anderson, D., Bestuzheva, K., Chen, W.-K., Eifler, L., Gasse, M., Gemander, P., Gleixner, A., Gottwald, L., Halbig, K., Hendel, G., Hojny, C., Koch, T., Le Bodic, P., Maher, S. J., Matter, F., Miltenberger, M., Mühmer, E., Müller, B., Pfetsch, M. E., Schlösser, F., Serrano, F., Shinano, Y., Tawfik, C., Vigerske, S., Wegscheider, F., Weninger, D., and Witzig, J. The SCIP Optimization Suite 7.0. Technical report, Optimization Online, March 2020. URL http://www.optimization-online.org/DB_HTML/2020/03/7705.html.

Goldberg, N. and Eckstein, J. Boosting classifiers with tightened l0-relaxation penalties. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pp. 383–390, 2010.

Goldberg, N. and Eckstein, J. Sparse weighted voting classifier selection and its linear programming relaxations. *Information Processing Letters*, 112(12):481–486, 2012.

Günlük, O., Kalagnanam, J., Menickelly, M., and Scheinberg, K. Optimal generalized decision trees via integer programming. *Preprint arXiv:1612.03225*, 2016.

Günlük, O., Kalagnanam, J., Menickelly, M., and Scheinberg, K. Optimal decision trees for categorical data via integer programming. *Preprint arXiv:1612.03225*, 2018.

Gurobi. Gurobi Optimizer. http://www.gurobi.com, 2020.

Leskovec, J. and Shawe-Taylor, J. Linear programming boosting for uneven datasets. In *Proceedings of the Twentieth International Conference on Machine Learning (ICML-2003)*, pp. 456–463, 2003.

Long, P. M. and Servedio, R. A. Random classification noise defeats all convex potential boosters. In *Proceedings of the 25th international conference on Machine learning*, pp. 608–615. ACM, 2008.

Long, P. M. and Servedio, R. A. Random classification noise defeats all convex potential boosters. *Machine learning*, 78(3): 287–304, 2010.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., et al. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

Pfetsch, M. E. Branch-and-cut for the maximum feasible subsystem problem. *SIAM J. Optim.*, 19(1):21–38, 2008.

Savickỳ, P., Klaschka, J., and Antoch, J. Optimal classification trees. In *COMPSTAT*, pp. 427–432. Springer, 2000.

Schapire, R. E. The boosting approach to machine learning: An overview. In *Nonlinear estimation and classification*, pp. 149–171. Springer, 2003.

Shalev-Shwartz, S. and Wexler, Y. Minimizing the maximal loss: How and why. In *Proceedings of the 32nd International Conference on Machine Learning*, 2016.

Tjeng, V., Xiao, K., and Tedrake, R. Evaluating robustness of neural networks with mixed integer programming. *Preprint arXiv:1711.07356*, 2017.

Verwer, S. and Zhang, Y. Learning optimal classification trees using a binary linear program formulation. In *33rd AAAI Conference on Artificial Intelligence*, 2019.

XPRESS. FICO Xpress Optimizer. https://www.fico.com/en/products/fico-xpress-optimization, 2020.

Zhu, J., Zou, H., Rosset, S., and Hastie, T. Multi-class AdaBoost. *Statistics and its Interface*, 2(3):349–360, 2009.