Truncated Interior Point Method for LP-Boost

Curtis Fox, Supervisors: Dr. Michael Friedlander, Dr. Yifan Sun

August 9, 2024

1 Introduction

This project involved using an interior point solver to solve the linear program present in the LP-boost algorithm. In the paper [1], the authors add an entropy regularizer (which is a logarithmic term) to the linear program in their LP-boost algorithm. This motivated the idea to try using an interior point (IP) method to solve the linear program, since IP methods also involve adding a logarithmic term to the objective function. To further improve the test error, we would force the solver to run for fewer iterations for earlier model fittings than for later model fittings (since boosting algorithms successively fit models, as we will describe in section 1.2). Thus, our goal was to use an interior point solver and to truncate the number of iterations of the solver to allow us to improve both the runtime and the test error of the boosting algorithm.

1.1 Classification Algorithms

A classification algorithm takes in an n by m matrix with n training examples and m features, as well as an n by 1 vector with the corresponding labels for each training example. The labels take on values $\{1, -1\}$ (but in some applications there can be more than just 2 labels). After the algorithm fits a model to our data, the hope is that this model allows us to predict accurate labels on our training data, as well as on additional data not used in training, known as test data.

1.2 Boosting

Boosting algorithms work by taking a weighted sum of weak learners (the weak learners are classification models) to make predictions on test data. Boosting algorithms successively fit models by using an update rule, which will vary based on the particular boosting algorithm. Note that a boosting algorithm has both a probability distribution d on the training examples, and a probability distribution w on the models fitted. Each time a new model is fitted, a new probability distribution is placed on the training examples, based on which examples were previously classified correctly or incorrectly. Predictions will take the form $y_j = \sum_{i=1}^{t} w_i m_i(x_j)$, where we are making a prediction on example j, and y_j is our predicted label. Using this notation there are t models fitted, where m_i is the ith model fitted by the boosting algorithm, and w_i is the weight of the ith model.

1.3 Linear Programming

The goal of linear programming is to find a value x that minimizes the optimization problem of the following form:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & c^T x \\ \text{subject to} \quad & Ax = b \\ & x \geq 0 \end{aligned} \tag{1}$$

Note that a linear program can be formulated in different ways, but the way we present it here is known as standard form for an LP. To find x such that this optimization problem is minimized, we can use various algorithms such as the simplex algorithm, as well as an interior point method.

2 Methodology

The first step in the project was implementing the LP-boost algorithm based on [1]. In our implementation of the algorithm, we did not implement a termination condition, but instead fitted a fixed number of weak learners.

Since a boosting algorithm involves taking a weighted average of weak learners, we needed to choose the kind of weak learner to use. We chose to use decision stumps (decisions trees with only one split). Decision stumps can be fitted more quickly and possibly lead to less overfitting than using deeper decision trees.

On each iteration of the boosting algorithm, we must fit a new distribution to our training examples. This distribution is found by solving a linear program.

The linear program, based on [1], has the form:

$$\begin{aligned} \underset{d,\gamma}{\text{minimize}} \quad & \gamma \\ \text{subject to} \quad & Ud \leq \gamma \\ & \sum_{j=1}^{n} d_j = 1 \\ & d \geq 0 \end{aligned} \tag{2}$$

The dual linear program, based again on [1], has the form:

$$\begin{aligned} \underset{w,\zeta}{\text{maximize}} \quad & \zeta \\ \text{subject to} \quad & U^T w \geq \zeta \\ & \sum_{i=1}^{t} w_i = 1 \\ & w \geq 0 \end{aligned} \tag{3}$$

U is a matrix of all 1's and -1's. Each row of this matrix corresponds to a fitted model, where $u_{ij} = 1$ indicates that the jth training example was correctly labeled by the model i, and $u_{ij} = -1$ indicates that the jth training example was incorrectly labeled by the model. Using the notation above, the final model has the form $y_j = \sum_{i=1}^{t} w_i m_i(x_j)$, where y_j is the predicted label for example j. Note that m_i is the ith decision stump fitted by the boosting algorithm, and $m_i(x_j)$ is the label assigned to x_j by m_i.

Unlike in the paper [1], in our formulation of the problem, we do not include a smoothness constraint $d \leq \frac{1}{v}e$, where v is a scalar such that $v \in [1, n]$ (n is the length of d, or the number of training examples), and e is a vector of ones of length n.

In the original implementation of the algorithm, we used CVX, as cited in [3], to solve the linear program associated with the algorithm. We then compared the training and test error using different forms of regularization (plots shown in results section).

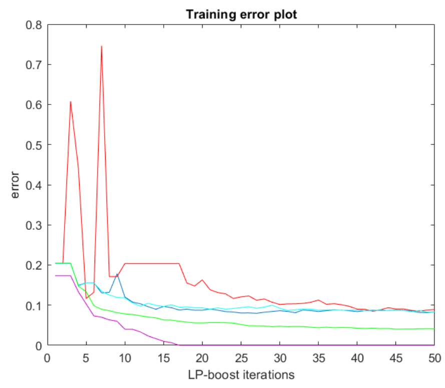
The final step was to set an upper bound on the number of iterations the interior point solver could run, which is sometimes referred to as truncation. We used the primal-dual interior point solver PDCO, as cited in [2] (note that the linear program (2) was first converted to standard form, since the solver is for equality-constrained problems). The reason we truncated the number of iterations of the solver was to attempt to improve both the runtime and the test error. By doing this, the solver will terminate more quickly, and will hopefully lead to less overfitting in earlier model fittings of the boosting algorithm. Various truncation schemes were used, including:

- Running at most i iterations of the solver for the ith model fitting (results shown in the next section)
- Running at most $\frac{i}{2}$ iterations of the solver for the ith model fitting
- Stepwise increase of the maximum number of solver iterations (for example, only increasing the number of iterations the solver can run every 25 model fittings)

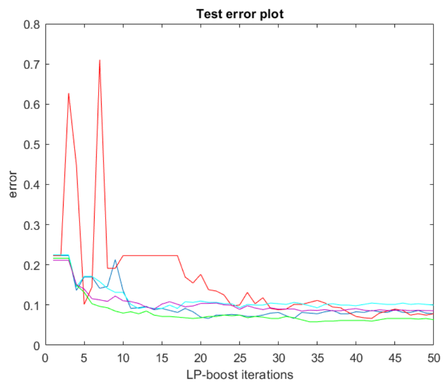
3 Results

In each of our simulations, we used approximately 80% of the data for training, and approximately 20% for testing. We used two datasets, spambase and waveform. The spambase dataset has 4601 examples and 57 features, where 22.6% of the matrix entries are nonzero. The waveform dataset has 5000 examples, and 21 features, where 99.8% of the matrix entries are nonzero. Both of these datasets are from the UCI Machine Learning Repository.

For the spambase dataset, we first ran a simulation without truncation. We allowed the solver to perform up to 50 primal-dual iterations on each model fitting. We then ran a simulation with truncation. The truncation scheme we used was as follows: For the ith iteration of the LP-boost algorithm, the solver used at most i primal-dual iterations. Both these simulations were done using PDCO. We also ran simulations using 1-norm regularization, 2-norm regularization, and entropy regularization. These 3 simulations were done using CVX. We plotted both the training and test error comparing all five methods:



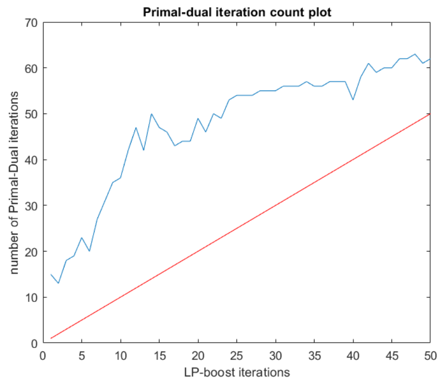
(a) Training Error plot



(b) Test Error plot

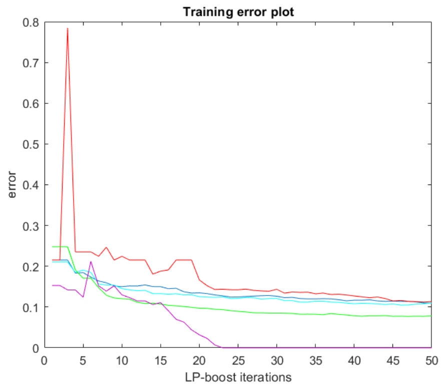
Figure 1: Plots for the spambase dataset, where the blue lines correspond to using no truncation, the red lines correspond to using truncation, the cyan lines correspond to 1-norm regularization, the green lines correspond to 2-norm regularization, and the purple lines correspond to entropy regularization

For the methods using PDCO, the time for the boosting algorithm to complete with no truncation was 19 seconds, and with truncation 12 seconds. The time spent solving the linear program (so not including time spent doing tasks such as fitting decisions stumps or predicting labels) with no truncation was 15 seconds, and with truncation 8 seconds. The following plot compares the number of iterations the PDCO solver used with and without truncation:

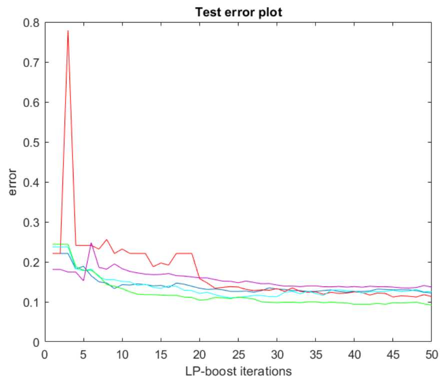


(a) PDCO Solver Iterations plot for the spambase dataset, where the blue line corresponds to using no truncation, and the red line correspond to using truncation

For the waveform dataset, we performed similar experiments to the spambase dataset, and produced the following training and test error plots:



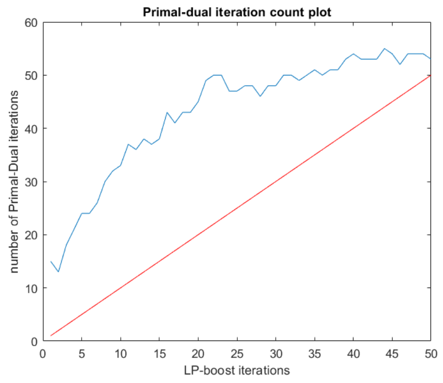
(a) Training Error plot



(b) Test Error plot

Figure 3: Plots for the waveform dataset, where the blue lines correspond to using no truncation, the red lines correspond to using truncation, the cyan lines correspond to 1-norm regularization, the green lines correspond to 2-norm regularization, and the purple lines correspond to entropy regularization

For the methods using PDCO, the time for the boosting algorithm to complete with no truncation was 19 seconds, and with truncation 14 seconds. The time spent solving the linear program (so not including time spent doing tasks such as fitting decisions stumps or predicting labels) with no truncation was 14 seconds, and with truncation 10 seconds. The following plot compares the number of iterations the PDCO solver used with and without truncation:



(a) PDCO Solver Iterations plot for the waveform dataset, where the blue line corresponds to using no truncation, and the red line correspond to using truncation

4 Conclusion

Overall, we were able to improve the runtime of the boosting algorithm, but not consistently improve the test error. None of the truncation schemes we used lead to any significant decrease in test error. Some additional experiments that someone could try include using regularization with an IP solver for the LP-boost algorithm rather than truncation, and see if this leads to significantly improved test error over using truncation. In addition, one could compare the results on sparse datasets versus dense datasets. In our experiments, we did not use sparse datasets, and it's possible that larger improvements in test error could be achieved using them. As well, we used decision stumps as our weak learners. Another idea would be to use a different kind of classification algorithm as the weak learner for the boosting algorithm.

References

[1] Warmuth M.K., Glocer K.A., Vishwanathan S.V.N. (2008) Entropy Regularized LPBoost. In: Freund Y., Györfi L., Turán G., Zeugmann T. (eds) Algorithmic Learning Theory. ALT 2008. Lecture Notes in Computer Science, vol 5254. Springer, Berlin, Heidelberg

[2] Saunders, Michael, PDCO, (2018), GitHub repository, https://github.com/mxsaunders/pdco

[3] Michael Grant and Stephen Boyd. CVX: Matlab software for disciplined convex programming, version 2.1. http://cvxr.com/cvx, December 2017.