Machine Learning with Applications 7 (2022) 100243

Contents lists available at ScienceDirect

Machine Learning with Applications

journal homepage: www.elsevier.com/locate/mlwa

AddGBoost: A gradient boosting-style algorithm based on strong learners

Moshe Sipper [a,*], Jason H. Moore [b]

[a] Department of Computer Science, Ben-Gurion University, Beer Sheva 84105, Israel
[b] Institute for Biomedical Informatics, University of Pennsylvania, Philadelphia, PA 19104-6021, USA

ARTICLE INFO

Keywords:
Gradient boosting
Regression

ABSTRACT

We present *AddGBoost*, a gradient boosting-style algorithm, wherein the decision tree is replaced by a succession of (possibly) stronger learners, which are optimized via a state-of-the-art hyperparameter optimizer. Through experiments over 90 regression datasets we show that AddGBoost emerges as the top performer for 33% (with 2 stages) up to 42% (with 5 stages) of the datasets, when compared with seven well-known machine-learning algorithms: KernelRidge, LassoLars, SGDRegressor, LinearSVR, DecisionTreeRegressor, HistGradientBoostingRegressor, and LGBMRegressor.

1. Introduction

In a seminal paper, "The strength of weak learnability", (Schapire, 1990) described a method "for converting a weak learning algorithm into one that achieves arbitrarily high accuracy". A weak learner is one that can produce an hypothesis that performs only slightly better than random guessing, while a strong learner can with high probability output an hypothesis that is correct on all but an arbitrarily small fraction of the instances. Over the years, a plethora of highly successful *boosting* algorithms that transform weak learners into strong ones have been devised, including AdaBoost (Freund & Schapire, 1997), gradient boosting (Friedman, 2001), XGBoost (Chen & Guestrin, 2016), and LightGBM (Ke et al., 2017).

It would seem that by and large the emphasis in boosting techniques has been on weak learners, typically decision trees. Works using strong learners in the context of boosting employed mainly AdaBoost-like (Freund & Schapire, 1997) boosting. Modest success was attained by Fink and Perona (2004), Harries (1999). Wickramaratna et al. (2001) showed that boosting a strong learner with AdaBoost may, in fact, contribute to performance degradation. Suggala et al. (2020) recently presented a boosting framework where, unlike traditional boosting, more complex forms of aggregation of weak learners were used.

Herein we focus on gradient boosting, which works by iteratively building prediction models, with each model attempting to minimize the error over the residual errors of the previous stage. Rather than use a weak learner (typically a decision tree) we deploy learners that are usually considered strong ones.

In the next section we describe our method, called AddGBoost. Section 3 presents the experimental setup and our results, followed by concluding remarks in Section 4.

2. AddGBoost

For our experiments we used the popular Python-based scikit-learn package, due to its superb ability to handle much of the desiderata of machine learning coding and experimentation (Pedregosa et al., 2011; *Scikit-learn: Machine Learning in Python*, 2020).

The main idea behind AddGBoost is simple: we replace the boosted weak learner of gradient boosting (typically a decision tree) with a succession of learners selected from the following seven algorithms (we use the scikit-learn and LightGBM function names for ease of reference):

1. `KernelRidge` (kernel ridge regression)
2. `LassoLars` (lasso model fit with least angle regression)
3. `SGDRegressor` (a linear model fitted by minimizing a regularized empirical loss with stochastic gradient descent)
4. `LinearSVR` (linear support vector regression)
5. `DecisionTreeRegressor` (a decision tree regressor)
6. `HistGradientBoostingRegressor` (histogram-based gradient boosting regression tree)
7. `LGBMRegressor` (LightGBM regressor)

Algorithm 1 provides the pseudocode (the code is available at https://github.com/moshesipper). AddGBoost receives a list of models to be used as boosting stages. Producing a final model is done in a standard gradient-boosting manner, through successive stages, where each stage fits a learner to the pseudo-residuals of the previous stage. Prediction is performed by summing up all learner predictions. The only change vis-a-vis standard gradient boosting involves the learners themselves, which are not decision trees but rather a mixture of the above seven algorithms.

* Corresponding author.
E-mail addresses: sipper@bgu.ac.il (M. Sipper), jhmoore@upenn.edu (J.H. Moore).

https://doi.org/10.1016/j.mlwa.2021.100243
Received 19 January 2021; Received in revised form 15 December 2021; Accepted 16 December 2021
Available online 22 December 2021

Algorithm 1 AddGBoost

Input:

models ← a list of models used as successive boosting stages

```
function FIT(X, y)   # X: training inputs, y: target values
    y_res = y
    for model in models do
        model.fit(X,y_res)   # fit current stage's model to pseudo-residuals
        y_res = y_res - model.predict(X)   # compute new pseudo-residuals

function PREDICT(X)   # X: inputs
    prediction = 0   # vector of zeros
    for model in models do
        prediction = prediction + model.predict(X)
    return prediction
```

Table 1
Parameter value ranges or sets used by Optuna.

Algorithm	Parameter	Values
`KernelRidge`	kernel	{'linear', 'poly', 'rbf', 'sigmoid'}
	alpha	[1e−4, 1]
	gamma	[0.01, 10]
`LassoLars`	alpha	[1e−04, 1]
`SGDRegressor`	alpha	[1e−05, 1]
	penalty	{'l2', 'l1', 'elasticnet'}
`LinearSVR`	C	[1e−05, 1]
	loss	{'epsilon_insensitive', 'squared_epsilon_insensitive'}
`DecisionTreeRegressor`	criterion	{'mse', 'friedman_mse', 'mae'}
	splitter	{'best', 'random'}
	max_features	{'sqrt', 'log2', None}
	min_samples_leaf	[1, 5]
`HistGradientBoostingRegressor`	max_iter	[10, 100]
	learning_rate	[0.01, 0.3]
	loss	{'least_squares', 'least_absolute_deviation'}
`LGBMRegressor`	lambda_l1	[1e−8, 10.0]
	lambda_l2	[1e−8, 10.0]
	num_leaves	[2, 256]

To obtain the models given as input to AddGBoost (Algorithm 1) we used Optuna, a state-of-the-art automatic hyperparameter optimization software framework (Akiba et al., 2019). Optuna offers a define-by-run-style user API where one can dynamically construct the search space, and an efficient sampling algorithm and pruning algorithm. Moreover, our experience has shown it to be fairly easy to set up. Optuna formulates the hyperparameter optimization problem as a process of minimizing or maximizing an objective function that takes a set of hyperparameters as an input and returns its (validation) score. We used the default Tree-structured Parzen Estimator (TPE) sampling algorithm. Optuna also provides pruning: automatic early stopping of unpromising trials (Akiba et al., 2019).

Optuna was tasked with performing the hyperparameter search through the space spanned by the values in Table 1, finding the best hyperparameters for a specific algorithm. Where AddGBoost is concerned, we added the seven regressors themselves into Optuna's mix—as part of the hyperparameter search space. To wit, Optuna was charged with finding *the best list of regressors* along with *each one's hyperparameters*.

For example, when tasked with optimizing the hyperparameters of an `SGDRegressor` for a given dataset using a specified number of trials (which we set to 100), Optuna will output the best hyperparameters, e.g.:

```
{'SGDRegressor_alpha': 0.005514232144543442,
     'SGDRegressor_penalty': 'elasticnet'}
```

When Optuna is tasked with optimizing the hyperparameters for `AddGBoost` with, say, 2 stages, the output takes the form:

```
{'regressor_0': 'SGDRegressor', 'SGDRegressor_alpha_0':
     0.17554186704629302, 'SGDRegressor_penalty_0': 'l2',
     'regressor_1': 'LGBMRegressor', 'LGBMRegressor_lambda_l1_1':
     0.00013890127227359438, 'LGBMRegressor_lambda_l2_1':
     6.62629152306247e-07, 'LGBMRegressor_num_leaves_1': 121}
```

Note that for fairness Optuna was given the same number of trials (100) as any other algorithm, though it was tasked with searching through a larger search space (which increases with the number of stages). Yet despite this prima facie disadvantage we shall see below that successful models were often obtained.

3. Experimental setup and results

We tested AddGBoost on 90 regression datasets from the PMLB repository (Orzechowski et al., 2018), focusing on those with a small number of instances (47–1000) and an intermediate number of features (2–124). This allowed us to perform a large number of runs over a sizeable number of datasets. Moreover, in many fields there is a need for good algorithms over small datasets (e.g., Inés et al. (2021)).

The pseudo-code for the experimental setup is given in Algorithm 2. For each dataset and for each of the seven algorithms above plus AddGBoost we performed 100 replicate runs, each with a 70%–30% random train-test split. We fit scikit-learn's `StandardScaler` to the training set and applied the fitted scaler to the test set. This ensured that features had zero mean and unit variance (often helpful for non-tree-based algorithms).

For each algorithm, including AddGBoost, Optuna found the best model over the training set, which was then tested on the test set. Note that where AddGBoost is concerned, 'model' refers to a succession of

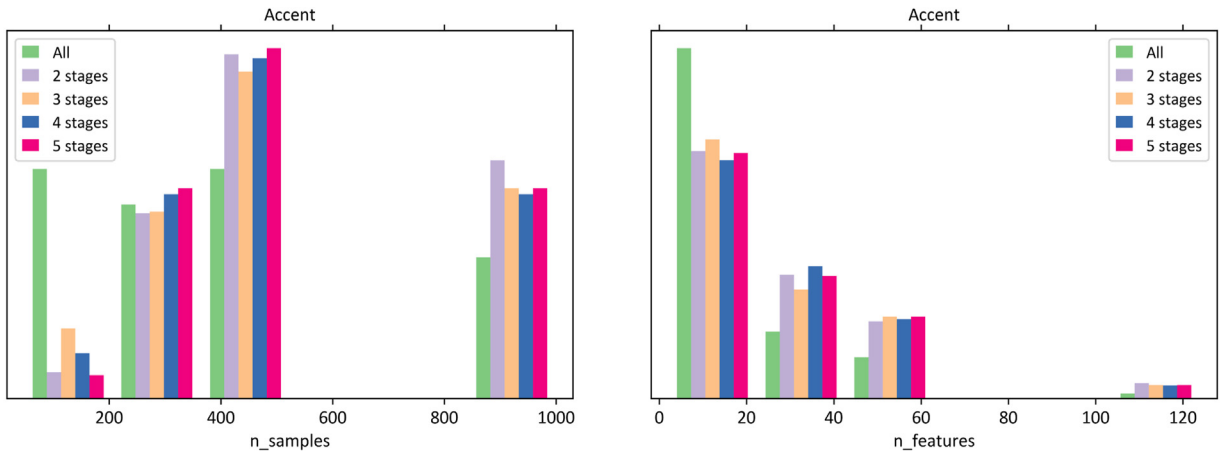

Fig. 1. Histograms of number of classes and number of features for all datasets vs. the datasets for which AddGBoost emerged as number 1. Note: histograms are normalized such that the integral over the range is 1 (specifically, each bin displays the bin's raw count divided by the total number of counts and the bin width).

Algorithm 2 Experimental setup (per dataset)

Input:

dataset ← dataset to be used
algorithms ← {`AddGBoost`, `KernelRidge`, `LassoLars`, `SGDRegressor`, `LinearSVR`, `DecisionTreeRegressor`, `HistGradientBoostingRegressor`, `LGBMRegressor`}

Output:

Test scores (for each algorithm over all replicates)

1: **for** *rep* ← 1 to *100* **do**
2: Randomly split *dataset* into 70% *training* set and 30% *test* set
3: Fit `StandardScaler` to *training* set and apply fitted scaler to *test* set
4: **for** *alg* in *algorithms* **do**
5: Run Optuna with *alg* for 100 trials over training set and obtain *best model*
6: Compute mean absolute error of *best model* on *test* set

Table 2
Results of 100 replicates per each of 90 datasets. Each replicate comprised running 8 algorithms—AddGBoost and the seven algorithms of Section 2. Stages: number of boosting stages used by AddGBoost. 1st: number of datasets for which AddGBoost was ranked number one. Sig: number of datasets for which AddGBoost's number-one rank was statistically significant (when compared with 2nd-place algorithm). 2nd: number of datasets for which AddGBoost was ranked number two. Insig: number of datasets for which AddGBoost's number-two rank was statistically insignificant (when compared with 1st-place algorithm).

Stages	1st	Sig	2nd	Insig
2	42	30	31	19
3	51	34	24	13
4	48	35	24	13
5	47	34	26	16

stages, where Optuna must find which regressor algorithm to use per each stage, and the hyperparameters of each one.

For each dataset we computed the median of the test scores of all 100 replicates. We then ranked the algorithms from best to worst. When AddGBoost took first place, we performed a 10,000-round permutation test, comparing the median scores of AddGBoost with the second-place algorithm. If the p-value was < 0.05 the win was considered significant. When AddGBoost came second, we performed a 10,000-round permutation test, comparing the median scores of AddGBoost with the first-place algorithm. If the p-value was $>= 0.05$ the second place was considered insignificant. We ran the experiments over all 90 datasets 4 times, with number of stages set to 2, 3, 4, and 5. Table 2 shows our results.

Fig. 1 shows histograms of number of classes and number of features for all datasets vs. the datasets for which AddGBoost emerged as number 1. We note that AddGBoost seems to perform better when the number of samples is higher, and when the number of features is higher.

Table 3
Results with 200 Optuna trials.

Stages	1st	Sig	2nd	Insig
2	49	30	27	18
3	53	34	20	11
4	53	34	21	16
5	56	38	21	15

Finally, we asked whether doubling the number of Optuna trials—from 100 to 200—would increase the success rate. Table 3 shows our results, evidencing that improvement, if any, was minor.

4. Concluding remarks

We presented AddGBoost, a gradient boosting-style algorithm, wherein the decision tree is replaced by a succession of stronger learners. The final model, composed of a list of models along with their hyperparameters, is found via a state-of-the-art hyperparameter optimizer. Our testing has shown that AddGBoost can improve results significantly for a substantial number of the datasets we tested. As can be seen in Tables 2 and 3, AddGBoost emerges as the statistically significant top performer for 33% (with 2 stages) up to 42% (with 5 stages) of the 90 datasets examined. Further, it attains a number-two, statistically insignificant second place for 12% to 21% of the datasets. As noted above, AddGBoost seems to perform better when the number of samples is higher, and when the number of features is higher. As an

added benefit, as can be seen in Algorithm 1, AddGBoost is quite easy to code.

There are a number of avenues we can offer for future exploration:

- AddGBoost can be deployed with algorithms other than those tested herein; indeed, one can even run AddGBoost with a single algorithm. This choice also influences runtime, which is almost entirely dependent on the runtimes of the underlying algorithms.
- Test AddGBoost with larger datasets and examine whether they bear out our findings that AddGBoost is advantageous when more samples and features are given.
- Add a scaling (weight) coefficient to successive boosting stages. This could be fixed or change dynamically.
- Our focus herein was on regression. It would seem worthwhile to examine AddGBoost for classification.
- While we focused on gradient boosting, other types of boosting techniques might be examined as to whether they might be a good fit for AddGBoost.

CRediT authorship contribution statement

Moshe Sipper: Conceptualization, Methodology, Software, Writing – original draft. **Jason H. Moore:** Data Analysis, Statistics, Investigation, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported by National Institutes of Health (USA) grants LM010098, LM012601, AI116794. We thank Hagai Ravid for spotting an error in an earlier version of the code.

References

Akiba, T., Sano, S., Yanase, T., Ohta, T., & Koyama, M. (2019). Optuna: A next-generation hyperparameter optimization framework, In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining* (pp. 2623–2631).

Chen, T., & Guestrin, C. (2016). XGBoost: A scalable tree boosting system. In *KDD: vol. 16, Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (pp. 785–794). New York, NY, USA: ACM, http://dx.doi.org/10.1145/2939672.2939785.

Fink, M., & Perona, P. (2004). Mutual boosting for contextual inference. In *Advances in neural information processing systems* (pp. 1515–1522).

Freund, Y., & Schapire, R. E. (1997). A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, *55*(1), 119–139.

Friedman, J. H. (2001). Greedy function approximation: a gradient boosting machine. *The Annals of Statistics*, 1189–1232.

Harries, M. B. (1999). Boosting a strong learner: Evidence against the minimum margin. In *ICML: vol. 99, Proceedings of the Sixteenth International Conference on Machine Learning* (pp. 171–180). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc..

Inés, A., Domínguez, C., Heras, J., Mata, E., & Pascual, V. (2021). Biomedical image classification made easier thanks to transfer and semi-supervised learning. *Computer Methods and Programs in Biomedicine*, *198*, Article 105782.

Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., Ye, Q., & Liu, T.-Y. (2017). Lightgbm: A highly efficient gradient boosting decision tree. In *NIPS: vol. 17, Proceedings of the 31st International Conference on Neural Information Processing Systems* (pp. 3149–3157). Red Hook, NY, USA: Curran Associates Inc..

Orzechowski, P., La Cava, W., & Moore, J. H. (2018). Where are we now? A large benchmark study of recent symbolic regression methods. In *Proceedings of the genetic and evolutionary computation conference* (pp. 1183–1190).

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., & Duchesnay, E. (2011). Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, *12*, 2825–2830.

Schapire, R. E. (1990). The strength of weak learnability. *Machine Learning*, *5*(2), 197–227.

Scikit-learn: Machine learning in python. (2020). https://scikit-learn.org/. (Accessed: 20 November 2020).

Suggala, A., Liu, B., & Ravikumar, P. (2020). Generalized boosting. *Advances in Neural Information Processing Systems*, *33*.

Wickramaratna, J., Holden, S., & Buxton, B. (2001). Performance degradation in boosting. In *International workshop on multiple classifier systems* (pp. 11–21). Springer.