Gradient Boosting With Piece-Wise Linear Regression Trees

1 Histograms for GBDT With PL Tree

Histogram is used in several GBDT implementations [Tyree *et al.*, 2011; Chen and Guestrin, 2016; Ke *et al.*, 2017] to reduce the number of potential split points. For each feature, a histogram of all its values in all data points is constructed. The boundaries of bins in the histogram are chosen to distribute the training data evenly over all bins. Each bin accumulates the statistics needed to calculate the loss reduction. When finding the optimal split point, we only consider the bin boundaries, instead of all unique feature values. After the histogram is constructed, we only need to record the bin number of each feature for each data point. Fewer than 256 bins in a histogram is enough to achieve good accuracy [Zhang *et al.*, 2017], thus a bin number can be stored in a single byte. We can discard the original feature values and only store the bin numbers during the boosting process. Thus using histograms produces small memory footprint.

It seems nature to directly use the histogram technique in our algorithm. For LightGBM and XGBoost, each bin in a histogram only needs to record the sum of gradients and hessians of data in that bin. For our algorithm, the statistics in the histogram is more complex. The statistics are used to compute the least squares. Each bin B needs to record both $\sum_{i\in B} h_i \mathbf{x}_i \mathbf{x}_i^T$ and $\sum_{i\in B} g_i \mathbf{x}_i$, where $\mathbf{x}_i$ is the column vector of selected regressors of data i. However, the feature values $\mathbf{x}_i$ are needed when fitting linear models in leaves. We still need to access the feature values constantly, which incurs long memory footprint. To overcome this problem, for each feature j and each bin i of j, we record the average feature values in bin j, denoted as $\overline{x}_{i,j}$. When fitting linear models, we use $\overline{x}_{i,j}$ to replace the original feature value $\mathbf{x}_{k,i}$. Here $\mathbf{x}_{k,i}$ is the value of feature i of data point $\mathbf{x}_k$, and $\mathbf{x}_{k,i}$ falls in bin j. In this way, we can still discard the original feature values after preprocessing. Thus we adapt the histogram technique to PL Trees and preserve the small memory footprint.

The histogram technique used in several existing methods (such as XGBoost and LightGBM) only record 2 elements in each bin (sum of gradients and hessians). Our histograms require more than 2 elements (including qudratic terms of feature values). So the detailed implementation is in fact quite different from existing ones. To better utilized the SIMD units when constructing histograms, we use intel intrinsics (which directly indicate the assemble instructions to use) to carefully arrange the calculation.

2 Fully Corrective Additive and Half-Additive Fitting

As we stated in Section 4.1 of the paper, suppose the linear model of leaf s is $f(X) = b + \sum_{j=1}^{m} \alpha_j X_j$. When we split s into s_1 and s_2 with condition $X_q \leq c$, the linear model of s_1 and s_2 is $f(X) = b' + \sum_{j=1}^{m} \alpha'_j X_j + \alpha'_{m+1} X_q$. Here $b', \alpha'_1, ..., \alpha'_m, \alpha'_{m+1}$ are parameters needed to be fit with samples in each child leaf, respectively. Here we clearly define the *fully corrective*, *additive* and *half-additive fitting*. But the cost grows with the number of features involved in the linear model.

fully corrective fitting: All the parameters $b', \alpha'_1, ..., \alpha'_m, \alpha'_{m+1}$ should be recalculated in each child leaf with equation (6) in our paper. This is our original algorithm, which in general can provide best accuracy.

additive fitting: Only parameters b' and α'_{m+1} will be recalculated using least square. And $\alpha'_1, ..., \alpha'_m$ will be equal $\alpha_1, ..., \alpha_m$. Here $\alpha_1, ..., \alpha_m$ are the fitted parameters of the parent node s. This is the approach proposed by [Friedman, 1979]. It is cheap but the accuracy will be much worse than fully corrective fitting.

half-additive fitting: To further speed up GBDT-PL, we propose half-additive fitting, which is a trade-off between the above 2 methods. In half-additive fitting, parameters b' and α'_{m+1} will be recalculated, and $\alpha'_1, ..., \alpha'_m$ will be $\alpha_1, ..., \alpha_m$ scaled by a parameter β'. So now the linear model becomes $f(X) = b' + \beta'(\sum_{j=1}^{m} \alpha_j X_j) + \alpha'_{m+1} X_{m+1}$. Here $\alpha_1, ..., \alpha_m$ are the fitted parameters of the parent node s. b', β' and α'_{m+1} will be calculated with least square. In other words, we take $\sum_{j=1}^{m} \alpha_j X_j$ as a single feature for the child leaves. So we use $X_1, ...X_m, X_{m+1}$ in the linear model at the cost of using only 3 features, since only 3 parameters needed to be calculated. This brings an important speedup for GBDT-PL, with a little sacrifice in accuracy, as we can see from Table 1 of Section 6.1 in the paper.

3 Experiment Platform for Training Time Recording

The experiment environment for training time comparison are listed in Table 1.

Table 1: Experiment Platform

OS	CPU	Memory
CentOS Linux 7	2 × Xeon E5-2690 v3	DDR4 2400Mhz, 128GB

4 Datasets

The datasets we used in this paper are all from UCI datasets. The number of instances and features can be found in the following table. We will provide details about how we split the datasets into train and testing sets on our github page. [1] For

Table 2: Datasets Description

name	# training	# testing	# features	task
HIGGS	10000000	500000	28	classification
HEPMASS	7000000	3500000	28	classification
CASP	30000	15731	9	regression
Epsilon	400000	100000	2000	classification
SUSY	4000000	1000000	18	classification
SGEMM	193280	48320	14	regression
SUPERCONDUCTOR	17008	4255	81	regression
CT	42941	10559	384	regression
Energy	15788	3947	27	regression
Year	412206	103139	90	regression

datasets with features of large values, including Year, SUPERCONDUCTOR, Energy and CASP, we first find the minimum and maximum values of each feature in the training set, and rescale the features into range [0, 1] before feeding it into GBDT-PL. This is for numerical stability when computing matrix inversions.

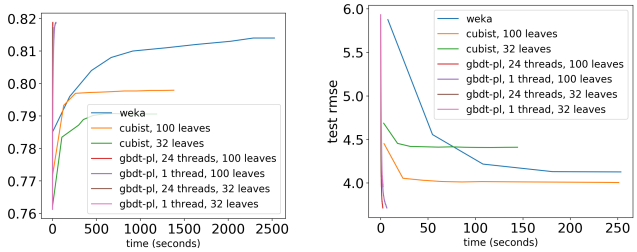
5 Comparison With Existing Boosted PL Trees

We compare our results with boosted PL Trees in Weka [Hall *et al.*, 2009] and Cubist [Kuhn *et al.*, 2018] packages. The base learner of Weka and Cubist is M5/M5P, a PL Tree proposed by [Quinlan and others, 1992; Wang and Witten, 1996]. The main differences between M5/M5P and our algorithm are: **1**. M5/M5P does not use half-additive fitting, histogram and our system optimization techniques. **2**. M5/M5P grows the tree in a way similar to piecewise constant regression trees (e.g. CART), then fits the linear models at the nodes. Each split in our algorithm considers how much the resultant linear models in the child nodes will reduce the boosting objective. In other words, the split finding in GBDT-PL is greedy and more expensive. Figure 1 shows the results on a small subset of HIGGS dataset with only 100k samples and CASP. All algorithms train 100 trees. GBDT-PL has a significant advantage in efficiency. For example, GBDT-PL finished training 100 32-leaf trees for CASP within 2 seconds, while Cubist takes about 150 seconds.

6 Parameters Tuning

The parameters we tried are listed in Table 2. For LightGBM and XGBoost, we also test *max_bin*=1023. For big datastes (HIGGS, Epsilon, SUSY and HEPMASS), we only test the learning rate 0.1. We evaluate all different combinations of

[1] https://github.com/GBDT-PL/GBDT-PL.git



(a) HIGGS-100k AUC (b) CASP RMSE

Figure 1: Comparison With Weka and Cubist

these values. On small datasets, 96 combinations of parameters are tested for GBDT-PL and CatBoost, and 144 combinations of parameters are tested for XGBoost and LightGBM. The number of trees (iterations) are chosen according to the learning rate, For XGBoost, LightGBM and GBDT-PL, 500 trees for *learning_rate* 0.1, 1000 trees for *learning_rate* 0.05 and 5000 trees for *learning_rate* 0.01. For CatBoost, we use 4 times of trees in other 3 packages. Since CatBoost uses only the first-order gradients, it convergences slower.

Table 3: Parameter Settings

num_leaves	16, 64, 256, 1024
max_bin	63, 255
min_sum_hessians	1.0, 100.0
learning_rate	0.01, 0.05, 0.1
l2_reg	0.01, 10.0

References

[Chen and Guestrin, 2016] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794. ACM, 2016.

[Friedman, 1979] Jerome H Friedman. A tree-structured approach to nonparametric multiple regression. *Smoothing techniques for curve estimation*, 757:5–22, 1979.

[Hall *et al.*, 2009] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[Ke *et al.*, 2017] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. In *Advances in Neural Information Processing Systems*, pages 3149–3157, 2017.

[Kuhn *et al.*, 2018] Max Kuhn, Steve Weston, Chris Keefer, and Maintainer Max Kuhn. Package 'cubist'. 2018.

[Quinlan and others, 1992] John R Quinlan et al. Learning with continuous classes. In *5th Australian joint confer-*

ence on artificial intelligence, volume 92, pages 343–348. World Scientific, 1992.

[Tyree *et al.*, 2011] Stephen Tyree, Kilian Q Weinberger, Kunal Agrawal, and Jennifer Paykin. Parallel boosted regression trees for web search ranking. In *Proceedings of the 20th international conference on World wide web*, pages 387–396. ACM, 2011.

[Wang and Witten, 1996] Yong Wang and Ian H Witten. Induction of model trees for predicting continuous classes. 1996.

[Zhang *et al.*, 2017] Huan Zhang, Si Si, and Cho-Jui Hsieh. Gpu-acceleration for large-scale tree boosting. *arXiv preprint arXiv:1706.08359*, 2017.