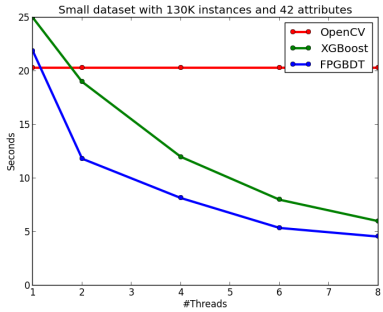
Parallel Gradient Boosting Decision Trees

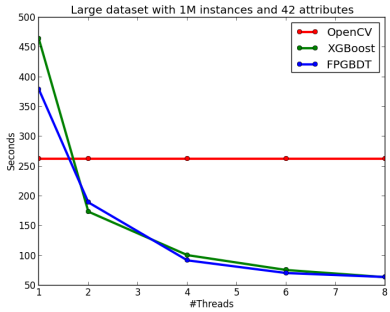
Zhanpeng Fang

zhanpenf@andrew.cmu.edu

Highlights

- Three different methods for parallel gradient boosting decision trees.
- My algorithm and implementation is competitve with (and in many cases better than) the implementation in OpenCV (http://opencv.org/) and XGBoost (https://github.com/dmlc/xgboost) (A parallel GBDT library with 750+ stars on GitHub).





Introduction

Gradient boosting is a machine learning technique for regression problems, which produces a prediction model in the form of an ensemble of weak prediction models. Gradient Boosting Decision Trees use decision tree as the weak prediction model in gradient boosting, and it is one of the most widely used learning algorithms in machine learning today. Its high accuracy makes that almost half of the machine learning contests are won by GBDT models. Below shows an example of the model.

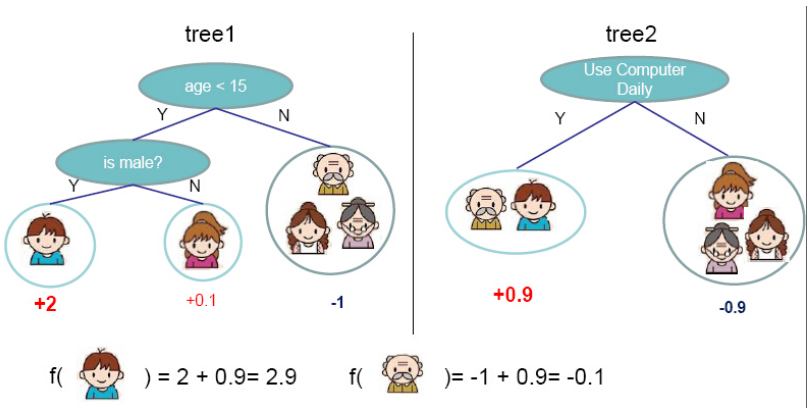


Figure from http://homes.cs.washington.edu/~tqchen/pdf/BoostedTree.pdf

The general idea of the method is *additive training*. At each iteration, a new tree learns the gradients of the residuals between the target values and the current predicted values, and then the algorithm conducts gradient descent based on the learned gradients. The algorithm description from Wikipedia is showed as followed:

Input: training set $\{(x_i, y_i)\}_{i=1}^{n}$, a differentiable loss function $L(y, F(x))$, number of iterations M.

Algorithm:

1. Initialize model with a constant value:
$$F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma).$$
2. For m = 1 to M:
    1. Compute so-called *pseudo-residuals*:
    $$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)} \quad \text{for } i = 1, \ldots, n.$$
    2. Fit a base learner $h_m(x)$ to pseudo-residuals, i.e. train it using the training set $\{(x_i, r_{im})\}_{i=1}^{n}$.
    3. Compute multiplier γ_m by solving the following one-dimensional optimization problem:
    $$\gamma_m = \arg\min_{\gamma} \sum_{i=1}^{n} L\left(y_i, F_{m-1}(x_i) + \gamma h_m(x_i)\right).$$
    4. Update the model:
    $$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x).$$
3. Output $F_M(x)$.

We can see that this is a sequential algorithm. Therefore, we can't parallelize the algorithm like Random Forest (http://en.wikipedia.org/wiki/Random_forest). We can only parallelize the algorithm in the tree building step. Therefore, the problem reduces to **parallel decision tree building**.

Experiment Setting

Since some experimental results will be showed when introducing the algorithm, we first introduce the dataset and the setting of the experiments. The data we use is from a competition of IJCAI'15 (http://ijcai-15.org/index.php/repeat-buyers-prediction-competition). We extracted one small dataset and one large datset from the data. The statistics of the small dataset and the large dataset are showed as follows:

- Small dataset: 130K instances, each with 42 attributes.
- Large dataset: 1M instances, each with 42 attributes, generated by duplicating the small dataset for eight times.

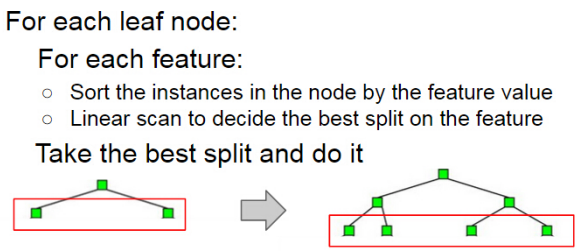
Without specification, the experimental results below are obtained from the small dataset. All the running time below are measured by growing 100 trees with maximum depth of a tree as 8 and minimum weight per node as 10. All the experiments are performed on a Debian machine with eight Intel E5-2650 2.0GHz cores and 64GB memory.

Sequential Decision Tree Building

We first define the input and output of the decision tree building problem:

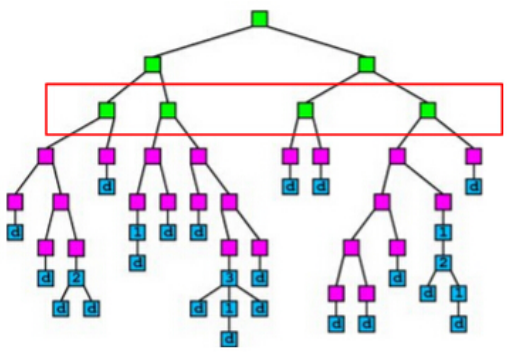
- **Input**: *N* instances each with *m* attributes and one target values.
- **Output**: A fitted decision tree for the input.

We also recap the algorithm for sequential decision tree building below. The building process grows a decision tree by levels. At each level, the algorithm first enumerates each leaf node at the level, and then conducts a split finding process on the node. During the split finding process for a node, the algorithm enumerates each features, then sorts the instances in the node by the feature values, and finds the best split of that feature in the node by linear scan. Finally the algorithm chooses the best split among those of all the features to split the node.

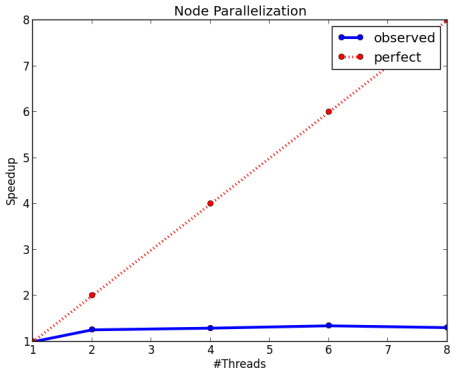


Method 1: Parallelize Node Building at Each Level

A simple idea of parallel decision tree bulding is to parallelize node building at each level. However, this method has a serious **workload imbalanced** problem. The reason is that a decsion tree tends to purify its nodes to obtain high prediction accuracy, and therefore many of the nodes will only contain a small group of training instances that have purified results, while some other nodes contain large group of trianing instances. The figure below shows an example of the imbalanced workload problem. Suppose we are going to build the nodes in the red box in parallel. We can see that the first and the third nodes contain much less training instances than the second and the fourth node, which causes the workload imbalanced.



The figure below shows the speedup of the node parallelization method. We can see that we only gain a very small speedup from node parallelization due to the workload imbalanced problem.

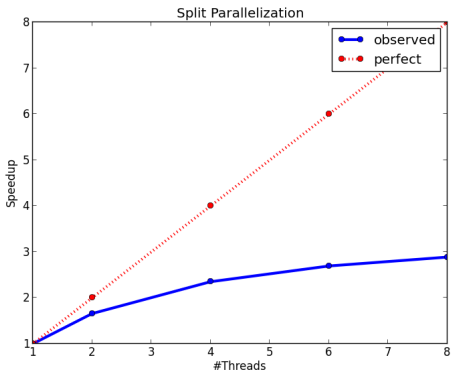


Method 2: Parallelize Split Finding on Each Node

Recall that in the split finding process on a node (the process is showed below), we need to enumerate each feature to find the split. The idea of this method is to parallelize the split finding process, so that in each node, the algorithm find split for different features in parallel.

1. For each feature:
   - Sort the instances by the feature value
   - Linear scan to decide the best split on the feature
2. Take the best split and do it

The speedup figure below shows that this method performs better than node parallelization. However, it still fails to achieve half of the peek speedup. The main problem of this method is that it will has **too much overhead for small nodes**. When a decision tree grows deeper, most of the nodes will only contain a small number of training instances. In this case, the computation cost for each node is very small, and the benefit brought by parallel computing can not cover the overhead brought by context switching, thread joining, and etc., which makes the method fails to achieve a good speedup. However, this method indeed points us to a correct direction, and our final method is based on parallel split finding by features.



Method 3: Parallelize Split Finding at Each Level by Features

It is showed above that at each level, the sequential building process of decision tree has two loops, the first one is an outer loop for enumerating the leaf nodes, and the second one is an inner loop that enumerates the features. The idea of this method is to swap the order of these two loops, so that we can parallelize the split finding for different features at the same level. A pseudocode of the algorithm is showed below. We can see that by changing the order of the loop, we also avoid sorting the instances in each node. We can sort the instances at the start of the whole building process, and then use the same sorting result at each level. On the other hand, note that to keep the correctness of the algorithm, each thread needs to carefully maintain their scaning status of each leaf node during the linear scan process, which significantly increases the coding complexity of the algorithm.

For each feature:
  For each leaf node:

Can sort globally first!

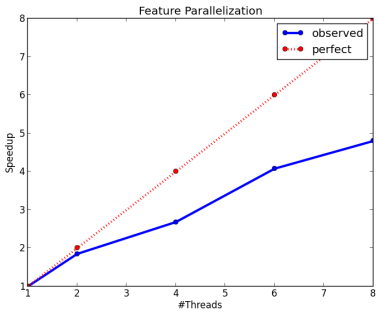
- ~~Sort the instances of the node by the feature value~~
- Linear scan to decide the best split on the feature

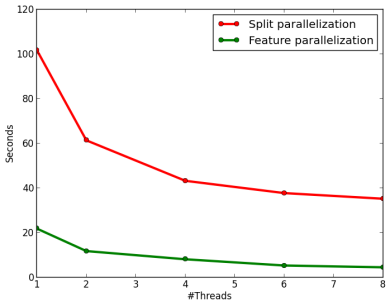
Take the best split for each node and do it

The advantages of the method are:

- **Workload are totally balanced.** Since the number of instances for each feature is the same, the workload for different jobs is the same. Thus, we do not have the workload imbalanced problem in method 1.
- **Overhead for parallelization is small.** Since we parallelize split finding at the whole level rather than a single node, the benefit from parallel computing is totally enough to cover the overhead from parallel computing.

The figures below show the speedup of the method and the running time comparison between this method and method 2. On the left figure, we can see that the method has an almost perfect speedup with two threads, and a 4.8x speedup with eight threads. On the right figure, we can see that the method is much faster than method 2, and it is because of the lower time complexity of the algorithm and the smaller overhead from multi-threading.



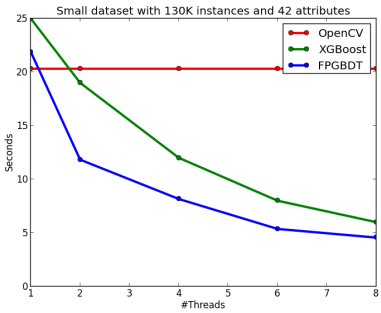


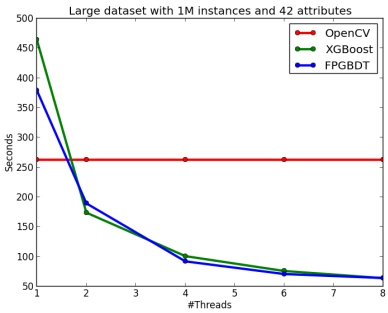
Compare with OpenCV and XGBoost

In this section, I show the effectiveness of my method and implementation by comparing with two strong baselines: OpenCV (http://opencv.org/) and XGBoost (https://github.com/dmlc/xgboost). The introduction of the two baselines are showed as follows:

- OpenCV: a sequential GBDT implementation in the OpenCV library. The algorithm will do early-stop pruning while growing a tree to reduce the computation.
- XGBoost: a parallel GBDT library with 750+ stars on GitHub.

My algorithm and implementation is competitve with (and in many cases better than) the implementation in OpenCV (http://opencv.org/) and XGBoost (https://github.com/dmlc/xgboost). The figures below show the running time of the three implementations on the two datasets. We can see that on the small dataset, my implementation is slightly slower than the OpenCV implementation on the single-thread case, but becomes faster than OpenCV when the number of threads is more than one since OpenCV does not have multi-thread implementation. When comparing with XGBoost on the small dataset, my implementation is faster than XGBoost on each tested thread configuration. On the large dataset, we can see that OpenCV performs significatly better in the single-thread case due to the early pruning technique, but worse than my method in the multi-thread cases. When comparing with XGBoost, in general, my implementation performs slightly better than XGBoost, but the performance difference between the two methods are not obvious.





[Code Download] (http://zhanpengfang.github.io/file_418/code.zip)

Last updated by Zhanpeng Fang, May. 11th 2015.