Tree Ensemble Learning on Parameter Server

Zhou Junhong
Department of Computer Science and Engineering
The Chinese University of Hong Kong
1155035315@link.cuhk.edu.hk

Abstract

In the last term, we studied and implemented gradient boosting tree in parameter server and compared with existing systems XGBoost and MLlib in Spark. Two preliminary problems found in our last research:

- For a given predict accuracy, our implementation takes more time or fails to achieve compared with XGBoost.
- Trees learned early are more significant and make the trees learned lately unimportant, thus it is hard to attain better accuracy with more trees.

Hence, in this term we introduce two solutions to relieve the issues mentioned above and improve our implementation: one is Candidate Splits Caching to speed up training process without losing accuracy significantly and the other is Tree Dropouts to make each tree trained contributes equally in the ensemble.

Also, the framework in project had been re-organized to provide a more general abstraction to implement decision tree algorithms on parameter server. Lastly, a channel module is implemented to support global communication among the machine nodes to provides a better scalability for gradient boosting tree on parameter server.

1 Background

1.1 Ensemble Learning

Ensemble learning is a machine learning paradigm where multiple learners are trained and then combined to predict new data points by taking a weighted vote of their predictions. In contract to ordinary machine learning approaches which try to learn one hypothesis from training data, ensemble methods try to construct a set of hypothesis and learn from them. In particular, tree ensemble is consisted of decision trees, each tree learns from residual from sum of predictions of previous trees and the labels. The original ensemble method is Bayesian averaging, but more recent algorithms include error-correcting output coding, Bagging and boosting.

1.2 Learning to Rank for Information Retrieval

Information Retrieval (IR). A process of locating and obtaining information that is needed by user. The modern IR works as follows: an information need of the user is defined by a query which is entered to the system, for example search query in a web search engine. Since the query is not unique identifier of any of the resources, it is possible that the query would match more than one resource. Moreover, each of the matched resources can show different degree of relevancy with respect to the given query. An IR algorithm is to find an appropriate ranking of the given documents (resources) according to their relatedness.

Learning to Rank (LTR). A method that can be effectively applied to solve the task of creating a ranking model in Information Retrieval. It is used to solve IR problems such as document retrieval, collaborative filtering, sentiment analysis, computational advertising etc. Given a training dataset of queries documents and evaluations of the relevancy among the documents, a LTR algorithm builds a ranking model, which is used to assign ranking scores to a new set (testing dataset) of document with unknown relevance later on. The evaluation of the performance of the model is accomplished by some performance metrics, such as Normalized Discounted Cumulative Gain (NDCG) and Expected Reciprocal Rank (ERR).

1.3 Dropouts

Dropouts is a tool that has been recently proposed in the context of learning deep neural networks (Hinton et al., 2012) to overcome the issue of over-specialization to considerable extent. In neural network, dropouts are used to mute a random fraction of the neural connections during the learning process. Hence, nodes at higher layers of the network cannot rely on a few connections to deliver the information needed for the prediction.

This method has contributed significantly to the success of deep neural networks for many tasks, for example, objective classification in images (Krizhevsky et al., 2012). It also has been used successfully in other learning models, for example in logistic regression (Wager et al., 2013) where dropouts are used to mute a random fraction of the input features during the training phase.

1.4 Related Work

There also have existing distributed systems or packages that implemented tree ensemble learning family and supported classification, regression and ranking on training dataset with decision trees.

RankLib (Dang, 2011). A Java library with several learning to rank algorithms implemented in Lemur Project[1], implemented algorithms include neural net based learning algorithms like LambdaRank and RankNet and tree based learning algorithms like MART and LambdaMART. Although its preliminary goal is not building machine learning algorithm in distributed system, this project borrows many good ideas in designing the package architecture and algorithms implementation from RankLib.

XGBoost (T. Chen, 2016). A scalable machine learning system for tree boosting. it provides parallel tree boosting (GBDT, GBM) that solve many data science problem and win in many data competition. The distributing strategy is AllReduce. Nodes are connected in tree-like style and exchange message. In below summation example, child nodes send value to parent nodes. Parent node sums up the values and send to the next parent nodes repeatedly. The root node has the final result and send back the result through the tree.

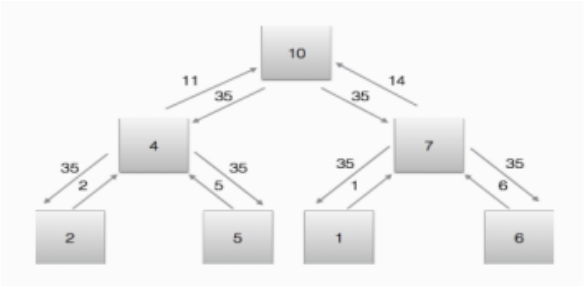


Fig 1. Summation example in AllReduce

LightGBM (G. Ke, 2017). It is also distributed system with gradient boosting framework that uses tree based learning algorithm and has been widely-used in many winning solution of machine learning competitions. The distributing strategy is Reduce-Scatter. In addition, LightGBM proposed two solutions to optimal the training process.

- Gradient-based One-Side Sampling (GOSS): it reserves all instances with large gradient values and does random sampling on instances with small instances. It showed that this approach can reduce computation cost significantly without losing much accuracy.
- Exclusive Feature Bundling (EFB): When dimension space of features is sparse, it is possible to bundle mutually exclusive features (i.e., they rarely take nonzero values simultaneously) to reduce the number of features without hurting the accuracy of split point determination by much.

Other Gradient Boosting Tree in Parameter Server. In addition to projects mentioned above, there have other distributed systems that implemented one or a few boosting tree algorithms in parameter server, namely GBDT in Angel from Tencent and LambdaMART in PSMART from Alibaba. Compare to these implementations, our package not only support both algorithms, but also provide more features (such as flexible objective functions and metrics) and flexibilities in building decision tree algorithms on parameter server.

2 Algorithms

2.1 Dropouts meet Multiple Additive Regression Trees (DART)

The idea of applying dropouts from neutral network to Multiple Additive Regression Tree (MART) is originated from Rashmi (2015). A common issue in MART is over-specialization: trees added at later iterations tend to impact the prediction of only a few instances, and make negligible contribution towards the prediction of all the remaining instances (A phenomenon appeared in our experiment in last term).

Shrinkage (Friedman, 2001, 2002) is the most common approach employed to relieve the issue of over-specialization. Since shrinkage reduces the impact of each tree by a constant value, the initial tree cannot compensate for the entire bias of the problem. However, it is found in experiment (Rashmi, 2015) that as the size of the ensemble increases, the problem of over-specialization reappears using shrinkage. Figure 2 shows the average contribution of the trees with the number of trees trained using different regularization approaches.

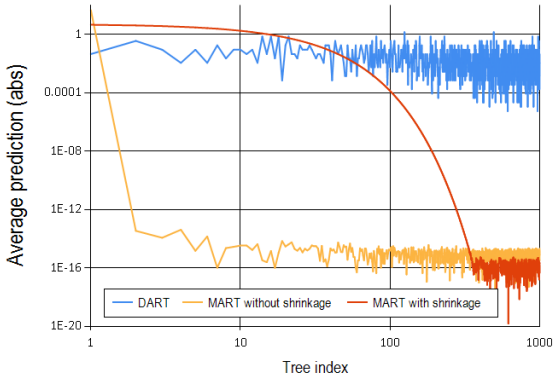


Fig 2. DART compares with Shrinkage (Rashmi, 2015)

The overall workflow of DART algorithm is similar to the GBDT algorithm we built in last term, but there have two modifications:

[1] http://www.lemurproject.org/

- When computing the gradient (estimator vector) that the next tree will fit, we will randomly drop trees with probability p_{drop} and use the rest of trained tree to update gradient.
- When adding new tree to the ensemble, DART will perform normalization on the new tree and the dropped trees. The rationale behind it is that the new trained tree T is trying to close the gap between current model and the true label, however the dropped trees are also trying to close the same gap. It will lead to the combined predicting model overshoot the target.

To normalize the new tree and the dropped trees, first assume that the number of dropped trees is k. Note that the new tree T has roughly k times larger magnitude than each of the dropped trees. Hence, we scale the new tree by a factor of $1/k$. Following this, now we have total (k + 1) which is more than k where we have not trained for new tree. Thus, we need to scale new tree and the dropped trees by a factor of $k/(k+1)$ to ensure that the combined effect remains the same as the effect of the dropped trees before introducing the new tree.

Algorithm: DART

Definitions:

N: total number of training trees

L'_x: first derivative of loss function with respect to estimated prediction on x.

$S_1 \leftarrow \{x, -L'_x(0)\}$

T_1 be a tree trained on the dataset S_1

$M \leftarrow \{T_1\}$

for t = 2 to N **do**

  D ← the subset of M such that T ∈ M is in D with probability p_{drop}

  if $D = \emptyset$ **then** D ← a random element from M

  end if

  $\widehat{M} \leftarrow M \setminus D$

  $S_t \leftarrow \{x, -L'_x(\widehat{M}(x))\}$

  T_t be a tree trained on the dataset S_t

  $M \leftarrow M \cup \{\frac{T_t}{|D|+1}\}$

  for T ∈ D **do**

    Multiply T in M by a factor of $\frac{|D|}{|D|+1}$

  end for

end for

Output M

2.2 LambdaMART

LambdaMART is a method combining MART and LambdaRank, where the design of gradient is originated from RankNet. Hence, we will begin by introducing RankNet.

RankNet

The common indicators of ranking, in general, cannot be applied to find gradient, thus it is impossible to perform gradient descending on ranking indicators. The innovation of RankNet is that, it converts ranking problem with un-derivable labels into an optimalization problem on probability of cross entropy loss function.

Given document x_i with scores s_i, we define a function that satisfies:

$$s_i = f(x_i, w)$$

Define the bias probability of two documents as

$$P_{ij} = P(x_i \rhd x_j) = \frac{1}{1 + \exp(-\sigma * (s_i - s_j))}$$

The loss function of cross entropy as

$$L_{ij} = -\overline{P_{ij}} log P_{ij} - (1 - \overline{P_{ij}}) \log(1 - \overline{P_{ij}})$$

where $\overline{P_{ij}}$ is the actual probability of x_i rank before x_j. Now we have gradient descending as

$$w_k \rightarrow w_k - \eta \frac{\partial L}{\partial w_k}$$

LambdaRank

In LambdaRank, it does factorization on $\frac{\partial L}{\partial w_k}$ to speed up the training process and eliminate the definition of loss function. It defines lambda on document i and j as:

$$\lambda_{ij} := \frac{\sigma}{1 + \exp(\sigma * (s_i - s_j))}$$

Based on this condition, consider and add change of rating indicator Z (such as NDCG):

$$\lambda_{ij} := \frac{\sigma}{1 + \exp(\sigma * (s_i - s_j))} * |\Delta Z_{ij}|$$

For a specific document i:

$$\lambda_i = \sum_{(i,j) \in P} \lambda_{ij} - \sum_{(j,i) \in P} \lambda_{ij}$$

where P is set of order pairs of all documents.

LambdaMART

Now we have the following situation: 1. MART offers a learning framework for LTR, but the gradient of ranking is missing; 2. LambdaRank defines a gradient λ_i for ranking. Thus, combining them, we have LambdaMART algorithm.

Algorithm: LambdaMART

Definitions:

F_i: tree model at iteration i.

N: number of trees

L: number of leave on tree

η: learning rate

for i : 0 → |{x, y}| **do**

  $F_0(x_i) = \text{BaseModel}(x_i)$

end for

for k = 0 to N **do**

  for i : 0 → |{x, y}| **do**

    $y_i = \lambda_i$

    $w_i = \frac{\partial y_i}{\partial F_{k-1}(x_i)}$

  end for

  $\{ R_{lk} \}_{l=1}^{L}$

  for l = 0 to L **do**

    $\gamma_{lk} = \sum_{x_i \epsilon R_{lk}} y_i / \sum_{x_i \epsilon R_{lk}} w_i$

  end for

  for i : 0 → |{x, y}| **do**

    $F_k(x_i) = F_{k-1}(x_i) + \eta \sum_l \gamma_{lk} I(x_i \in R_{lk})$

  end for

end for

Data Partition Strategy. Note that the training data is composed of pairwise examples for each query group, which may cause very high communication cost when the examples are irrationally distributed across workers. To avoid this problem, two methods (J. Zhou, PSMART, 2017) can be applied to divide the training data appropriately.

- Multiway number partitioning algorithm: Divide the query ids into different workers. Then, samples with the same query id are assigned to the same worker where the id belongs to.
- Approximate method: Store the samples of the same query id continuously in the file system. Then each worker loads evenly divided data as usual. If samples with the same query id are assigned into two workers, they will be treated as samples with two different query ids (could lose some pairwise samples).

2.3 Information Retrieval Measures

Information retrieval researchers use ranking quality measures such as Mean Reciprocal Rank (MRR), Mean Average Precision (MAP), Expected Reciprocal Rank (ERR), and Normalized Discounted Cumulative Gain (NDCG). NDCG (K. Jarvelin, 2000) and ERR (O. Chapelle, 2009) have the advantage that they handle multiple levels of relevance (whereas MRR and MAP are designed for binary relevance levels), and that the measure includes a position dependence for results shown to the user (that gives higher ranked results more weight), which is particularly appropriate for web search. NDCG is defined as follows. The DCG (Discounted Cumulative Gain) for a given set of search results (for a given query) is

$$\text{DCG@T} = \sum_{i=1}^{T} \frac{2^{l_i} - 1}{\log(1 + i)}$$

where T is the truncation level (for example, if we only care about the first page of returned results, we might take T = 10), and l_i is the label of the i-th document. We typically use five levels of relevance: $l_i \in \{0,1,2,3,4\}$. The NDCG is the normalized version of DCG:

$$\text{NDCG@T} = \frac{\text{DCG@T}}{\text{idealDCG@T}}$$

where the denominator is the maximum DCG@T attainable for that query, so that NDCG@T ∈ [0,1].

ERR was introduced more recently and is inspired by cascade models, where a user is assumed to read down the list of returned documents until they find one they like. ERR is defined as

$$\text{ERR} = \sum_{r=1}^{n} \frac{1}{r} R_r \prod_{i=1}^{r-1} (1 - R_i)$$

where, if l_m is the maximum label value,

$$R_i = \frac{2^{l_i} - 1}{2^{l_m}}$$

R_i models the probability that the user finds the document at rank position i relevant.

3 System Design

After re-organizing the framework, the project contains 4,000+ codes and supports user to run or create tree ensemble learning algorithm on parameter server by using a few API or codes. Most of the low-level logics such as how to train tree and config parameter server

for tree are transparent to user. The source code is available at https://github.com/Zizinc/flexps.

3.1 Architecture

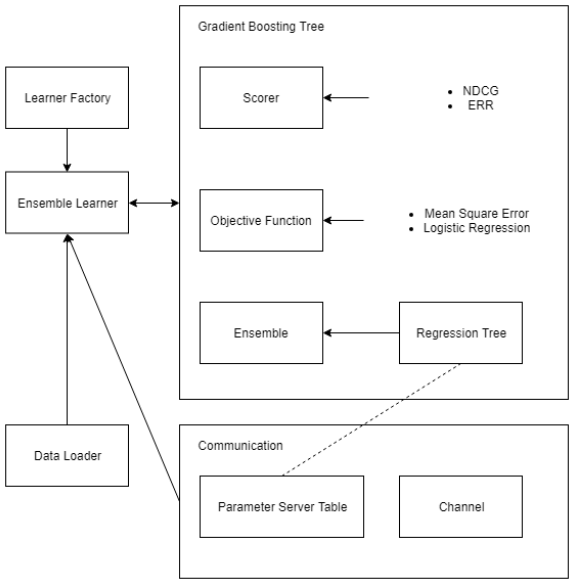


Fig 3. Architecture of Tree Ensemble Learning on Parameter Server

The above diagram illustrates the idea of implementing general tree ensemble learning algorithm on parameter server. Each small block represents the C++ class or abstraction.

Learner Factory. A factory class used to create learner. The learner must be declared in LearnerType.hpp in advance.

Ensemble Learner. An generic ensemble learner interface. The concrete learning algorithm need to inherit this class and implement methods train() and evaluate(). It also responsible for fetching training and test dataset and kv table for gradient boosting tree.

Currently GBDT, DART and LambdaMART are implemented.

Data Loader. This class responsible for loading and processing dataset. It supports two modes: loading from local machine (single node) or from HDFS. Data will be splitted equally and stored distributively for each thread before training with the algorithm. Each worker will fetch a portion of the dataset according to its node id and worker id. Other features include:

- Support loading data with SVM-Lib or SVM-Rank format.
- Return max and min of each feature in dataset.
- Split the dataset by feature during the training process
- Create new DataLoader according to worker id
- Support push and pop row data for workload balancing phase

Scorer. A generic retrieval measure computation interface. The concrete scoring metric needs to inherit this class and implement method calculate_score().

Some prevailing metrics are implemented, such as NDCG and ERR.

Objective Function (Loss Function). A generic objective function interface. The concrete objective function needs to inherit this class and implement method get_gradient().

Some possible functions are mean square error and logistic regression.

Ensemble. This class is responsible for managing the trained trees and support simple manipulations on them such as add_tree() and get_tree(). In particular, predict() in this class is to call predict() in each of the trained trees and return the aggregated result.

Regression Tree. The core class in the whole ensemble learner building process. It is responsible for training and predicting on the dataset. The detailed implementing logics of training tree are listed below:

- find_candidate_splits(): First echo push_quantile_sketch() to push local histogram result to parameter server. Later, pull the global result and echo find_candidate_split() to find candidate split points for the features.
- find_best_candidate_split(): First echo push_local_grad_hess() to push local information gain statistics to parameter server. Later pull the global result and echo find_best_split() to find the best splitting point on this node.
- find_predict_val(): When check_to_stop() returns true, it indicates that the current node is leaf. The program will push and pull from parameter server to find the predicting value on this leaf.
- reset_kv_tables(): After finishing training a node, the parameter server tables need to be reset for training the next node.
- train_child(): When check_to_stop() returns false, the program will assign and split dataset accordingly for left child and right child. Next, left_child.train() and right_child.train() are echoed and train recursively.
- update_leafs(): Scale the predictions on leaf by a constant factor (used for dropouts).
- information_caching(): Used to pass the quantile sketch, histogram and candidate split results to child to reduce communication and computation cost.

- predict(): Return the predicting value with shrinkage on this tree for a given data row.

Parameter Server Table. This class is responsible for creating and retrieving kv tables for regression tree. Thus, user can push and pull from tables directly without bothering to config parameter server table.

Channel. This class provides global communication among the nodes and support global operations such as workload balancing, finding max and min of features.

3.2 Programming Interface

Based on the architecture designed above, our program provides an easy-to-use interface for user to run gradient boosting tree algorithm or write new algorithm.

For running algorithm, user only need to load data to DataLoader, register kv tables at ParameterServer and create and config learner from LearnerFactory. Then, learner.train() and learner.evaluate() will do all the training and testing.

Programming interface: Run algorithm

```
1:  data_loader←Load from datasource
2:  parameter_server.create_kv_tables_for_tree_al
    go();
3:  task.SetLambda() {
4:      gbdt = learner_factory-> create_learner;
5:      gbdt->init(parameter_server.get_tables());
6:      params ← Config parameters // Such as
    number of trees, max depth
7:      gbdt->set_params(params);
8:      options ← Config options // Such as loss
    function, metrics
9:      gbdt->set_options(options);
10:     gbdt->set_data_loader(data_loader);
11:     gbdt->learn();
12:     predict_result = gbdt->evaluate();
13: }
```

For writing algorithm, user needs to implement two methods: learn() and evaluate().

- learn(): Unpack data from DataLoader, define estimator vector, create tree and call tree.train() and finally add to ensemble. Note that the details of training tree such as creating candidate histograms, finding information gain are transparent to user.
- evaluate(): Unpack data from DataLoader, ask ensemble to predict and finally use scorer to evaluate the predicting performance.

Programming interface: Write algorithm

```
1:  learn() {
2:      vects←Unpack data from DataLoader
3:      estimator_vect←Initialize estimator vector
4:      for t in num_of_trees:
5:          grad_vect = get_gradient_vect(vects,
    estimator_vect);
6:          RegressionTree tree;
7:          tree.init(vects, grad_vect, params);
8:          tree.set_kv_tables(kv_tables);
9:          tree.train();
10:         update_estimator_vect();
11:         ensemble.add_tree(tree);
12:     endfor
13: }
14:
15: evaluate() {
16:     map<string, float> predict_result;
17:     for vect in test_dataset:
18:         predict = ensemble.predict(vect);
19:         score = scorer.calculate_score(predict,
    vect);
20:         predict_result←Add score
21:     endfor
22: return predict_result;
23: }
```

3.3 Candidate Splits Caching

The idea of split candidates caching is to reuse the candidate splits from parent node during the tree training instead of re-calculating candidate splits for all features.

One important observation in training process is that given two features f_1 and f_2 that they are independent and identically distributed (i.i.d.) (the correlation coefficient of them is near to 0), if f_1 is chosen to be the splitting feature in parent node, the candidate splits of f_2 in parent node is similar to that in child node. Consider the following example.

Feature values for 3 different features:

f_1	1	1	1	2	2	9	9	10	10	10
f_2	1	1	1	3	3	4	4	5	5	5
f_3	1	3	4	5	2	4	1	5	3	2

Possible splits for this 3 features:

f_1	1.5	6.5	9.5	
f_2	2	3.5	4.5	
f_3	1.5	2.5	3.5	4.5

Suppose 6.5 in f_1 is selected to be the split point, in the left child we have:

f_1	1	1	1	2	2
f_2	1	1	1	3	3
f_3	1	3	4	5	2

The possible splits for left child:

f_1	1.5			
f_2	2			
f_3	1.5	2.5	3.5	4.5

We can notice that since f_3 is i.i.d. to f_1, the split candidates remain the same as the one in parent node. However, for f_2 which is highly correlated to f_1, the split candidates are altered.

Now consider f_1, f_2 and f_3 are large dataset and the above properties remain unchanged, we identify candidate splits by percentile. Since f_3 is i.i.d. to f_1, the upper bound, lower bound and data distribution tendency remain the same for f_3 in both parent node and child node. We can cache the candidate splits information from parent node and reuse for child nodes. On the other hand, it is possible that exists case like f_2 which is not i.i.d. to f_1, thus we also need to recompute the candidate splits for it.

Algorithm: Find candidate splits in the next tree node

Definitions:

f_{prev}: feature used to split in parent node

F: features set

C: candidate splits list

k: number of features to recompute

1: S ← pick top k feature f ∈ F in terms of correlation with f_{prev}

2: $C[f_{prev}] = find\_candidate\_splits(f_{prev})$

3: **for** f in S **do**

4: $C[f] = find\_candidate\_splits(f)$

5: **end for**

6: **for** f in F \ S **do**

7: $C[f] = use\_parent\_candidate\_splits(f)$

8: **end for**

Output C

3.4 Channel

One limitation in XGBoost is that it only supports distributed machine learning in single machine with multiple threads, but not support multiple machines mechanism like Hadoop or Spark. To overcome this, we provide a channel module for global communication among multiple machines in gradient boosting tree algorithm.

First, one node will be elected to aggregate information from all the nodes. The elected node will then schedule tasks to nodes or send the aggregated global result to nodes. After receiving instructions from elected node, nodes will execute the instruction and communicate with other nodes directly.

Workload Balancing. When loading data from HDFS, it is possible that some nodes will receive none or few data which leads to execution fails or inefficient parallelism. Channel module allows nodes to share data and make data loaded in each node as equal as possible. The workflow as below:

- Each node sends the data number loaded in local machine to the elected node.
- Elected node aggregates the data number and calculates the average number should be loaded on each machine.
- Elected node makes schedule to let nodes with number of data above average (resource available) allocates data to nodes with number of data below average (resource needed).
- Elected node distributes instructions to resource available nodes. Resource available nodes use DataLoader.pop() to dump data and send to destination nodes.
- Resource needed nodes receive data from resource available nodes directly and use DataLoader.push() to load data to local.

Find Global Max and Min on Features. Since each node only loads a portion of the data, they need to exchange information with each other to find the global max and min. First each node finds the max and min on each feature locally. Next, they send the local result to elected node by channel module. Elected node compares all the local results and then distributes the global max and min on each feature to all the nodes.

Find Global Predicting Result. Since each node only has a local predict result based on local data, they need

to share result in order to get a global predicting result. The workflow is similar to find global max and min on features.

4 Experiment

Dataset	Record Num	Feature Num	Format
MQ2008	136.9k	46	Rank-SVM
YearPredictionMSD	463.7k	90	Lib-SVM

Table 1: Dataset information

We use root-mean-square error (RMSE) to measure the accuracy of training result for YearPredictionMSD. Lower RMSE value means that the ML model more fits to the dataset. The formula is given as below:

$$\text{RMSE} = \sqrt{\frac{\sum_{i=1}^{n}(\hat{y}_i - y_i)^2}{n}}$$

where $\hat{y}_i$ is the evaluate result, y_i is the actual result and n is the number of records.

4.1 Comparison with Previous Implementation

This experiment is to check the performance of new implementation with candidate splits caching. As mentioned in previous section, candidate splits caching can help reduce communication and computation cost by reusing cached results from previous node (only part of features need to recompute for candidate splits). Hence, we shall expect the time spent will decrease and the predicting accuracy will decrease.

We use YearPredictionMSD dataset. The parameters of GBDT are set to:

- Number of tree: 20
- Max depth per tree: 6

Total Time Spent

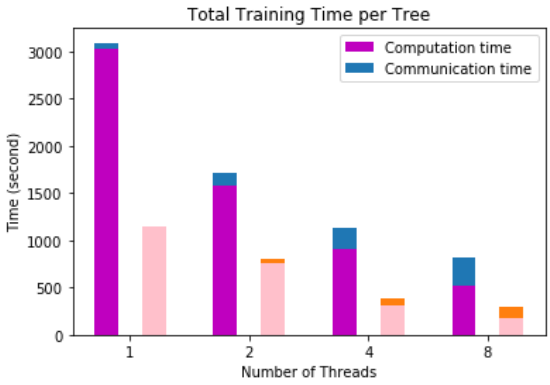


Fig. 4: Old version (left) v.s. new version (right) in terms of training time

For each thread 1, 2, 4 and 8, the left column (purple and blue color) is the previous implementation and the right column (pink and orange color) is the new implementation with candidate splits caching.

Threads	1	2	4	8
Previous version	3020s	1709s	1137s	811s
New version	1151s	800s	381s	296s
Improvement	62%	53%	66%	64%

Table: Total time spent on two versions

Threads	1	2	4	8
Previous version	3020s	1574s	910s	513s
New version	1151s	763s	316s	174s
Improvement	62%	52%	65%	66%

Table: Computation time spent on two versions

Threads	1	2	4	8
Previous version	0s	134s	227s	198s
New version	0s	34s	66s	120s
Improvement	0%	75%	71%	39%

Table: Communication time spent on two versions

We can notice from above figures that the new version speeds up the training process by 61% on average for different threads number, whereas computation time improves by 61% on average and communication time improves by 62% on average.

Predicting Accuracy

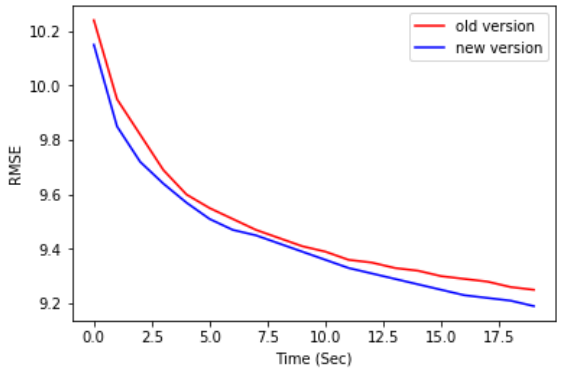


Fig 5. Old version v.s. new version in terms of training accuracy

On the other hand, surprisingly, we find that the new version still retains the training accuracy as the old version (or even supress slightly, however the result should be varied on different dataset). Therefore, it shows that using candidate splits caching not only can boost training time significantly, but also able to attain a valid predicting result.

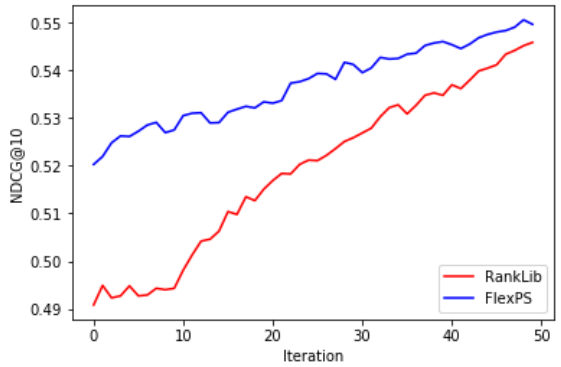
4.2 LambdaMART

This experiment compares the performance of LambdaMART on parameter server and that on RankLib in terms of NDCG@10 and ERR@10.

We use MQ2008 dataset. The parameters of LambdaMART are set to:

- Number of tree: 50
- Number of leave: 10

NDCG



ERR

	ERR@10
RankLib	0.0993
Our Parameter Server	0.0877

We can note from above figures that, during training process, our program consistently outperform LambdaMART in RankLib in term of NDCG by slight advantage. However, during testing process, our ERR@10 is lower than that of RankLib.

This may due to the effect of candidate caching mentioned above, as we skip to recompute split candidates of some of the features. Hence, we may overlook some candidate splits that is truly suitable for the entire dataset.

5 Conclusion

In this paper, we introduce a general framework to implement tree ensemble learning algorithms on parameter server. Several recent and popular gradient boosting tree algorithms (GBDT, LambdaMART, DART) are implemented in this project and demonstrates the flexibility and reusability of this framework.

We also mentioned several approaches, such as caching candidate splits from parameter servers and dropouts on trained tree to avoid over-specification, to fine tune the performance of training process of trees on parameter server and offer trade-off between training efficiency and training accuracy.

6 Acknowledgement

I would like to thank below people in this project:

- Prof. James Cheng, project supervisor, for offering research opportunity and kindly supervision.
- Yuzhen Huang, PhD student of James and author of FlexPS, for guidance and many suggestions in this project.

7 References

[1] Yuzhen Huang, Tatiana Jin, Yidi Wu, Zhenkun Cai, Xiao Yan, Fan Yang, Jinfeng Li, Yuying Guo, James Cheng. FlexPS: Flexible Parallelism Control in Parameter Server Architecture. PVLDB, 11(5): 5-57

[2] T.G. Dietterich. Ensemble methods in machine learning. Multiple classifier systems, Springer (2000), pp. 1–15.

[3] Tao Qin et al. LETOR: A benchmark collection for research on learning to rank for information retrieval. Springer Science+Business Media, LLC 2009.

[4] Georey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. Improving neural networks by preventing coadaptation of feature detectors. arXiv preprint arXiv:1207.0580, 2012.

[5] Rashmi Korlakai Vinayak, Ran Gilad-Bachrach. DART: Dropouts meet Multiple Additive Regression Trees. JMLR

[6] Tianqi Chen and Carlos Guestrin. XGBoost: A Scalable Tree Boosting System. In 22nd SIGKDD Conference on Knowledge Discovery and Data Mining, 2016

[7] 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 (NIPS), pp. 3149-3157. 2017.

[8] Jie Jiang, Jiawei Jiang, Bin Cui and Ce Zhang. TencentBoost: A Gradient Boosting Tree System with Parameter Server. ICDE, 2017

[9] Jie Jiang, Lele Yu, Jiawei Jiang, Yuhong Liu and Bin Cui. Angel: a new large-scale machine learning system. National Science Review (NSR), 2017

[10] Zhou, J., et al. PSMART: parameter server based multiple additive regression trees system. In: WWW 2017 Companion (2017)

[11] Mu Li, "Scaling Distributed Machine Learning with the Parameter Server" in OSDI, 2014.

[12] Q. Ho, J. Cipar, H. Cui, S. Lee, J. Kim, P. Gibbons, G. Gibson, G. Ganger, and E. Xing. "More effective distributed ml via a stale synchronous parallel parameter server" in NIPS, 2013.

[13] J. Wei, W. Dai et al. Managed communication and consistency for fast data-parallel iterative analytics. In SoCC, 2015

[14] Angel. https://github.com/tencent/angel.

[15] Apache Spark. https://spark.apache.org/.

[16] LightGBM. https://github.com/Microsoft/LightGBM.

[17] XGBoost. https://github.com/dmlc/xgboost.

[18] Jeffrey Dean and Sanjay Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters." December 2004, http://labs.google.com/papers/mapreduce.html

[19] Rajeev Thakur, Rolf Rabenseifner et al. "Optimization of collective communication operations in MPICH". The International Journal of High Performance Computing Applications, 2005.

[20] K. Jarvelin and J. Kekalainen. IR evaluation methods for retrieving highly relevant documents. Special Interest Group on Information Retrieval (SIGIR), 2000.

[21] O. Chapelle, D. Metzler, Y. Zhang and P. Grinspan. Expected Reciprocal Rank for Graded Relevance Measures. International Conference on Information and Knowledge Management (CIKM), 2009.

[22] C. Burges. From ranknet to lambdarank to lambdamart: An overview. Learning, 11:23{581, 2010.