Gradient Boosted Decision Trees on Hadoop

jerry ye  |  jyh-herng chow  |  jiang chen  |  zhaohui zheng
Agenda

- Overview
  - GBDT
  - Implementations
  - Related Work

- GBDT
  - Learning a tree
  - Boosting

- Method
  - MapReduce Implementations
  - MPI Implementation

- Results

- Conclusion
Gradient Boosted Decision Trees (GBDT) is a machine learning algorithm that iteratively constructs an ensemble of weak decision tree learners through boosting.
What is GBDT?

- Gradient Boosted Decision Trees was introduced by Jerome Friedman in 1999
- An additive regression model over an ensemble of trees, fitted to current residuals, gradients of the loss function, in a forward step-wise manner
- Favors many shallow trees (e.g., 6 nodes, 2000 trees)
- Advanced Algorithms: GBRank, SmoothDCG
- Numerous applications within Yahoo!
- Blender in Bellkor’s winning Netflix solution
Advantages

- Feature normalization is not required
- Feature selection is inherently performed during the learning process
- Not prone to collinear/identical features
- Models are relatively easy to interpret
- Easy to specify different loss functions
Disadvantages

- Boosting is a sequential process, not parallelizable
- Compute intensive
- Can perform poorly on high dimensional sparse data, e.g. bag of words
Known Implementations

- Salford’s TreeNet
- gbm package in R
- PLANET: Massively Parallel Learning of Tree Ensembles with MapReduce, Panda et. al.
- Tong Zhang implemented GBDT while at YRL
- More implementations at this workshop
Algorithm Overview
Algorithm

Algorithm:

\[ F_0(x) = \arg \min_{\gamma} \sum_{i=1}^{N} \Psi(y_i, \gamma) \]

For \( m = 1 \) to \( M \) do:

\[ y_{im} = -\left[ \frac{\partial \Psi(y_i, F(x_i))}{\partial \Psi(F(x_i))} \right]_{F(x) = F_{m-1}(x)}, \quad i = 1, N \]

\[ \{R_{lm}\}_1^L = \text{growtree}(\{y_{im}, x_i\}_1^N) \]

\[ \gamma_{lm} = \arg \min_{\gamma} \sum_{x_i \in R_{lm}} \Psi(y_i, F_{m-1}(x_i) + \gamma) \]

\[ F_m(x) = F_{m-1}(x) + \nu \cdot \sum_{l=1}^{L} \gamma_{lm} 1(x \in R_{lm}) \]

end

Friedman “Stochastic Gradient Boosting”, 1999

\{y,x\}: dataset
\Psi: loss function
\gamma: node score
M: number of trees
N: training set size
\nu: shrinkage
GBDT Process

- Each tree, partition sample space by growing n nodes
- Compute gradient and repeat
Splitting a node

Find the best split using Information Gain:

- Compute gain for each cut point
- Choose cut with highest gain
- Valid cutpoints:

\[
\begin{align*}
 f_1 &< 1.5 \quad \text{gain} = 10 \\
 f_1 &< 2.5 \quad \text{gain} = 30 \\
 f_1 &< 3.5 \quad \text{gain} = 57 \\
 f_2 &< 0.5 \quad \text{gain} = 25
\end{align*}
\]
Partitioning

Using best split, partition the data
- samples above the cut goes to left node
- samples below cut goes to right node
- find best cuts for new nodes

\[ f_1 < 3.5 \quad \text{gain} = 57 \]

residual' += mean(target_{node})
Boosting

\[ F_m(x) = F_{m-1}(x) + \nu \cdot \sum_{l=1}^{L} \gamma_{lm} 1(x \in R_{lm}) \]

\[ \text{gradient}_{m+1}(x) = \text{label}(x) - F_m(x) \]

• Gradient can vary depending on loss function, least squares shown
• Gradients are targets for next tree
• Stochastic boosting randomly subsamples training data for each tree
Example Tree

- ftsc < 0.99
  - ftsc < -1.82
    - fprx < 0.09
      - fpgprodrev < 6.5
        - fcdgst < 126.5
      - fquen0 < 0.5
    - ftsc < 4.06
      - favedurh180 < 83.5
        - ftsc < 5.98
        - fcdgst < 126.5
MapReduce Implementations
Horizontal: Finding Cuts

Partition Data:
- Each mapper emits (<feature, value>, <residual, weight>) pairs
- Reducers aggregates pairs and sorts
- Process scales as more nodes are added
Finding Splits

Each mapper emits:

- \((\text{<feature, value>}, \text{<residual, weight>})\)

Reducer aggregates cuts and sorts

Output: Sorted list of candidate cutpoints

Single pass over sorted list to compute best split

---

**Algorithm 1** Aggregating candidate splits

```plaintext
map(key, value):
    F ← set of features
    sample ← split(value, delim)
    for f in F do
        key = (f, sample[f])
        value = (sample[residual], sample[weight])
        emit(key, value)
    end for

reduce(key, values):
    residual_sum ← 0
    weight_sum ← 0
    for v in values do
        residual_sum ← residual_sum + v.residual
        weight_sum ← weight_sum + v.weight
    end for
    emit(key, (residual_sum, weight_sum))
```
Partitioning Data

• Split data according to cut
• Output to DFS

Algorithm 2 Partitioning a Node $n$

```plaintext
map(key, value):
sample ← split(value, delim)
if sample[n.feature] < n.splitpoint then
    residual = sample[residual]+ n.left_response
else
    residual = sample[residual]+ n.right_response
end if
emit(key, value)
```
MapReduce (horizontal)

- Scales with more mappers
- Method is slow!
- 5 minutes to train 1 node
- Takes 211 minutes to train a 63 node tree on 1.2M x 500 feature dataset
- Reading from HDFS can take 1-2 minutes. We have 3 MapReduce jobs for a tree node
- Keep information in memory – vertical partitioning
**Vertical: Partition Data**

- Each mapper gets a subset of features
- Read features into memory
- Mappers are persistent until ensemble is trained

<table>
<thead>
<tr>
<th></th>
<th>( f_0 )</th>
<th>( f_1 )</th>
<th>( f_2 )</th>
<th>( f_3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s_0 )</td>
<td>( V_{0,0} )</td>
<td>( V_{0,1} )</td>
<td>( V_{0,2} )</td>
<td>( V_{0,3} )</td>
</tr>
<tr>
<td>( s_1 )</td>
<td>( V_{1,0} )</td>
<td>( V_{1,1} )</td>
<td>( V_{1,2} )</td>
<td>( V_{1,3} )</td>
</tr>
<tr>
<td>( s_2 )</td>
<td>( V_{2,0} )</td>
<td>( V_{2,1} )</td>
<td>( V_{2,2} )</td>
<td>( V_{2,3} )</td>
</tr>
<tr>
<td>( s_3 )</td>
<td>( V_{3,0} )</td>
<td>( V_{3,1} )</td>
<td>( V_{3,2} )</td>
<td>( V_{3,3} )</td>
</tr>
<tr>
<td>( s_4 )</td>
<td>( V_{4,0} )</td>
<td>( V_{4,1} )</td>
<td>( V_{4,2} )</td>
<td>( V_{4,3} )</td>
</tr>
<tr>
<td>( s_5 )</td>
<td>( V_{5,0} )</td>
<td>( V_{5,1} )</td>
<td>( V_{5,2} )</td>
<td>( V_{5,3} )</td>
</tr>
</tbody>
</table>
Vertical Mappers

Mapper 1
- f3
- f4
- f5

Find best local cuts

partition

• f3<2.5

Output residuals

Update residuals

Mapper 2
- f6
- f7
- f8

Find best local cuts

partition

Find best local cuts

Update residuals

NFS

Wait for all cuts

Start

start

• f3
• f4
• f5

• f6
• f7
• f8
MPI Implementation
Message Passing Interface

- Message Passing Interface (MPI) allows many computers to communicate with each other.
- Dominant model in high performance computing
- Scalable, portable
- Distributed shared memory for high RAM jobs
- OpenMPI is an open source implementation of MPI
- Low level and can be complicated to use
- Modified OpenMPI to run on Hadoop
- Fault tolerance
Splitting a node

Each machine gets a feature
- Machine 1 finds local best split on $f_1$
- Machine 2 finds local best split on $f_2$
- Use MPI to broadcast local splits
- Best global split found

\[
\begin{align*}
    f_1 &< 1.5 \quad \text{gain} = 10 \\
    f_1 &< 2.5 \quad \text{gain} = 30 \\
    f_1 &< 3.5 \quad \text{gain} = 57 \\
    f_2 &< 0.5 \quad \text{gain} = 25
\end{align*}
\]

Global Best: $f_1 < 3.5$
Using best cut, split the data

- Only Machine 1 has $f_1$ in memory, partition dataset
- Partition is maintained in indices, send updated index to others
- All machines updates residuals

\[
f_1 < 3.5 \quad \text{gain} = 57
\]
Boosting

\[ F_m(x) = F_{m-1}(x) + \nu \cdot \sum_{l=1}^{L} \gamma_{lm} 1(x \in R_{lm}) \]

\[ \text{gradient}_{m+1}(x) = \text{label}(x) - F_m(x) \]

- Scores are kept for all samples through training of trees
- All machines computes new gradients and updates targets for next tree
- Repeat until finished
Experiments
Scalability

MPI implementation faster than MapReduce using vertical partitioning
Scalability

Scalability for different dataset sizes
Experiment

MapReduce

Horizontal: 211 minutes x 2500 trees = 366 days x 100 machines
Vertical: 28 seconds x 2500 trees = 19.4 hours x 20 machines

MPI

5 seconds x 2500 trees = 3.4 hours x 10 machines

1800% less node hours!
Application

- Dataset (2M doc, 600 features)
- Tree Parameters: typical
  - Trees(2500), Terminal Nodes(20)
- Running time: (Runtime Memory: 4GB)
  - Single thread, single machine: 7 days
  - Multi-threads (6), single machine: 3.5 days
  - MPI on grid: 9 hours with 20 nodes, 12 hours with 10 nodes
  - More complex loss: 16 days -> 36 hours
Conclusions

- We have implemented a distributed version of GBDT
- Distributed version running faster than sequential version
- Can handle larger datasets that sequential version cannot
- Advanced algorithms based on GBDT can benefit from this framework
- Implementation GBDT uses MPI on Hadoop
- GBDT 6X faster than achievable using MapReduce
- 1800% reduction in node hours
Thanks!

For more info: jerryye@yahoo-inc.com
References

1. AMDAHL, G. Validity of the single processor approach to achieving large-scale computing capabilities. pp. 483–485.


5. FOUNDATION, A. Apache hadoop project. lucene.apache.org/hadoop.


