Sequoia Forest

A Scalable Random Forest Implementation on Spark

Sung Hwan Chung, Alpine Data Labs

Random Forest Overview

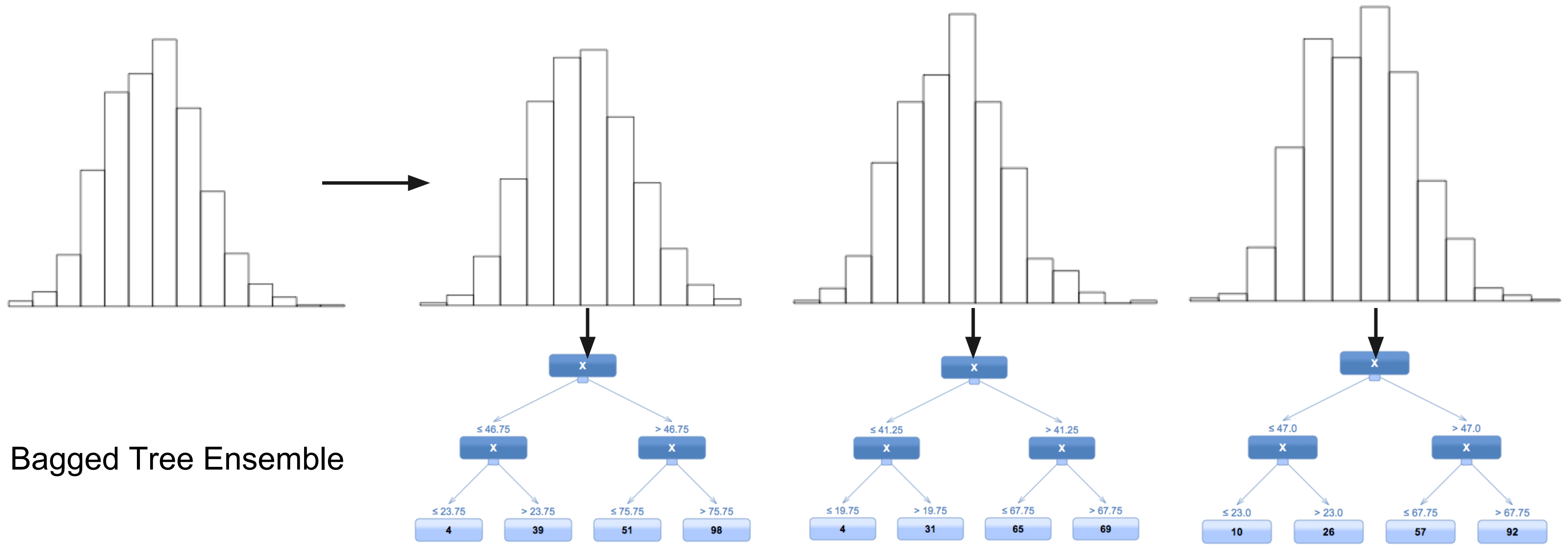
- It’s a technique to reduce the variance of single decision tree predictions by averaging the predictions of many decorrelated trees.
- The decorrelation of many decision trees is achieved through *Bagging* and/or randomly selecting features per tree node.

Bagging

Training Data

Bootstrap Samples

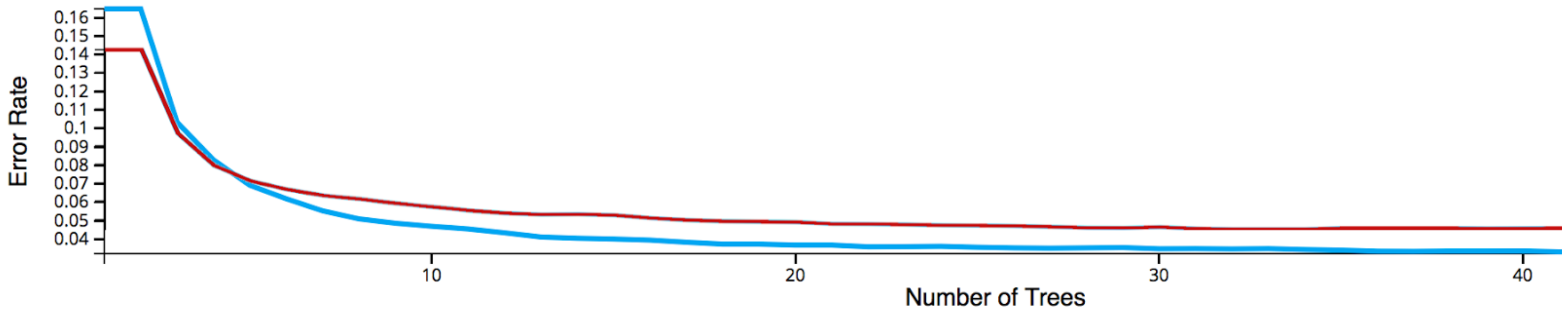
Bagged Tree Ensemble



Random Features Per Node

- Only examine a smaller random set of features.
- E.g., if there are 10000 features, randomly select 100 features for each node and determine optimal split based on these.
- This decorrelates trees more and often works better than bagging.

Bagging vs Random Features



MNIST data set

Red : Bagging, No Random Features

Blue : No Bagging, Random Features

Random Forest in Spark

- The main challenge is to train multiple individual decision trees over distributed data.
- Additionally, in Random Forest, decision trees are typically fully grown. So they get large particularly for a large data set.

Random Forest in Spark Contd.

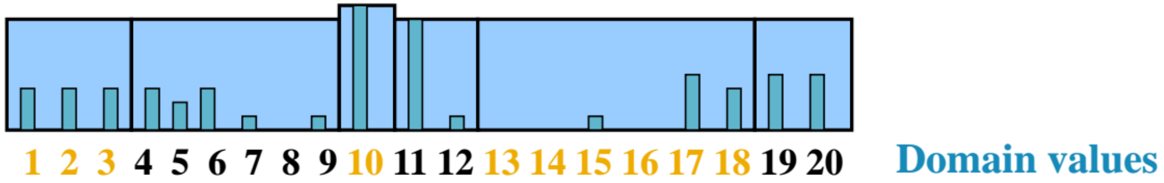
- ***Mllib*** already has a decision tree implementation but it doesn’t support random features and is not optimized for training a large number of nodes.
- We implemented our own to fit our needs.

Distributed Tree Training

- The big idea is similar to Google’s ***PLANET*** implementation (also adopted by ***Mllib***).
- Individual trees are built node by node in the driver node.
- At each iteration, individual executors collect partition statistics that are required to determine node splits and predictions.

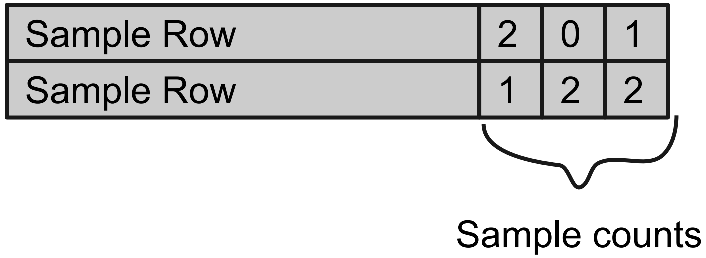
Preprocessing - Feature Discretization

- For efficiency, features are discretized first.



Preprocessing - Bagging

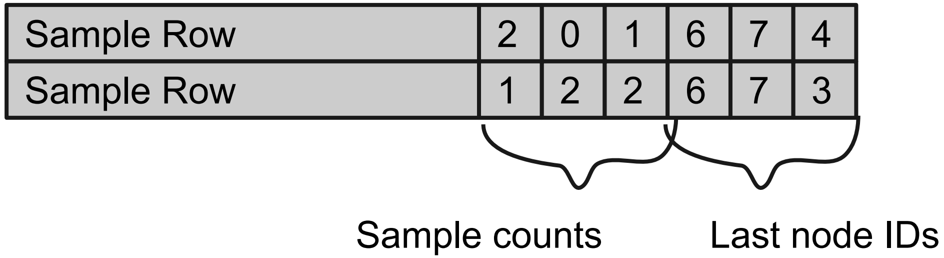
- Approximate random sampling - with/without replacement. executors tag rows with sample counts. One column per tree.
  - Bernoulli sampling for without-replacement.
  - Poisson sampling for with-replacement.



Sample Row	2	0	1
Sample Row	1	2	2

Mutable Node ID Tag

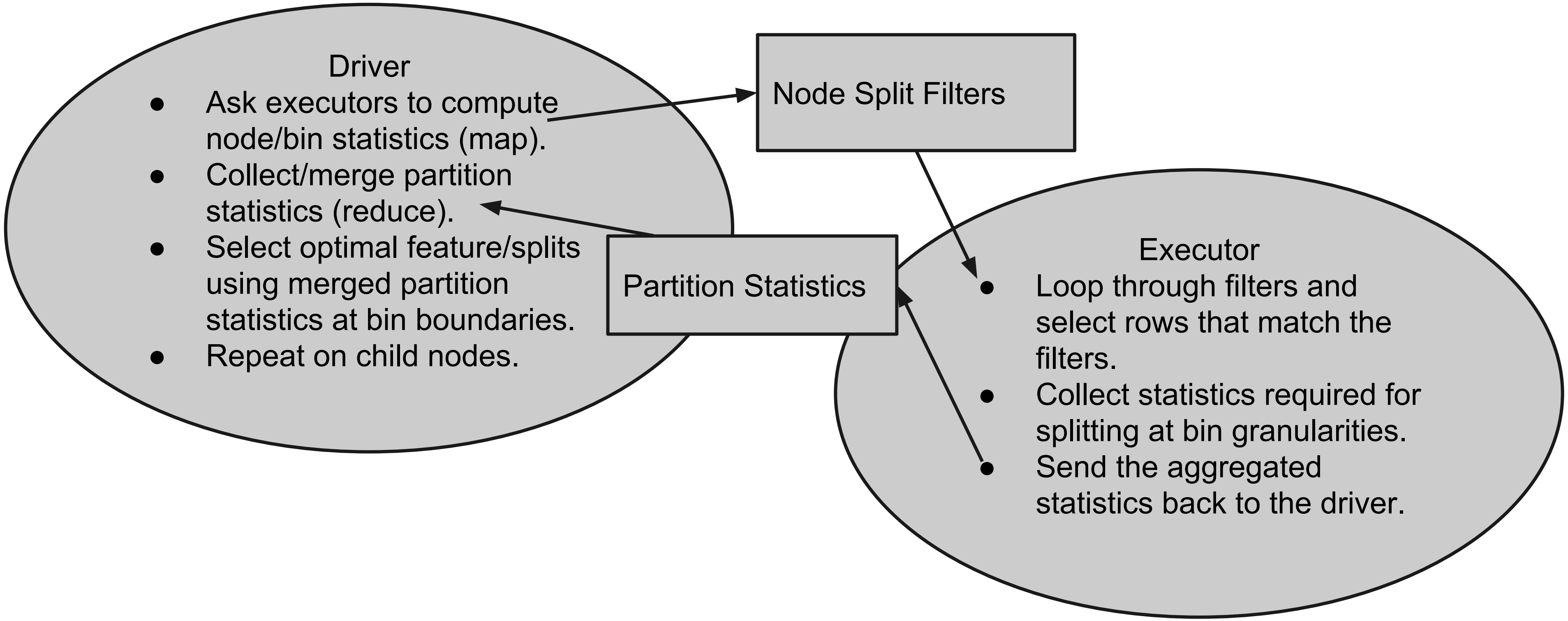
- Remember the last node that the row belonged to. One column per tree again.



Sample Row	2	0	1	6	7	4
Sample Row	1	2	2	6	7	3

- This speeds up the process considerably since we don’t have to pass large filters/trees to executors.

Training/Splitting Nodes



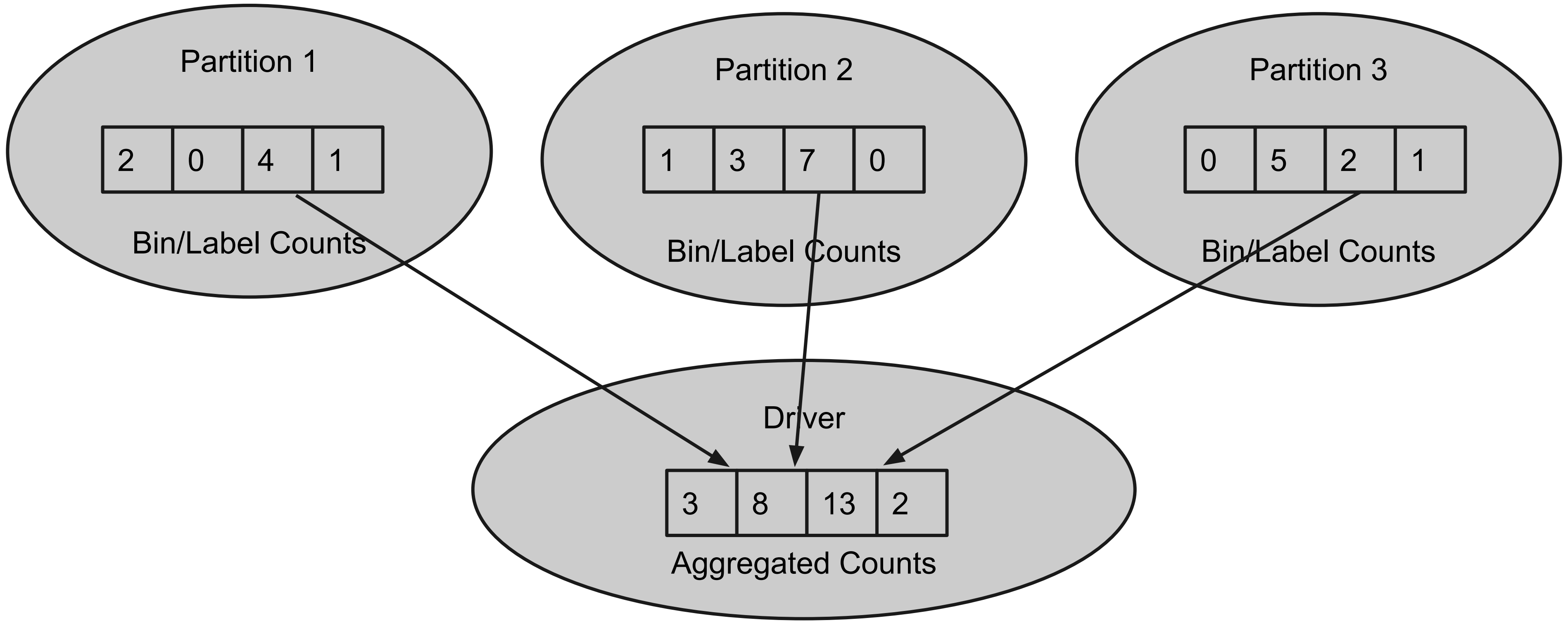
Driver

- Ask executors to compute node/bin statistics (map).
- Collect/merge partition statistics (reduce).
- Select optimal feature/splits using merged partition statistics at bin boundaries.
- Repeat on child nodes.

Executor

- Loop through filters and select rows that match the filters.
- Collect statistics required for splitting at bin granularities.
- Send the aggregated statistics back to the driver.

Partition Statistics Aggregation



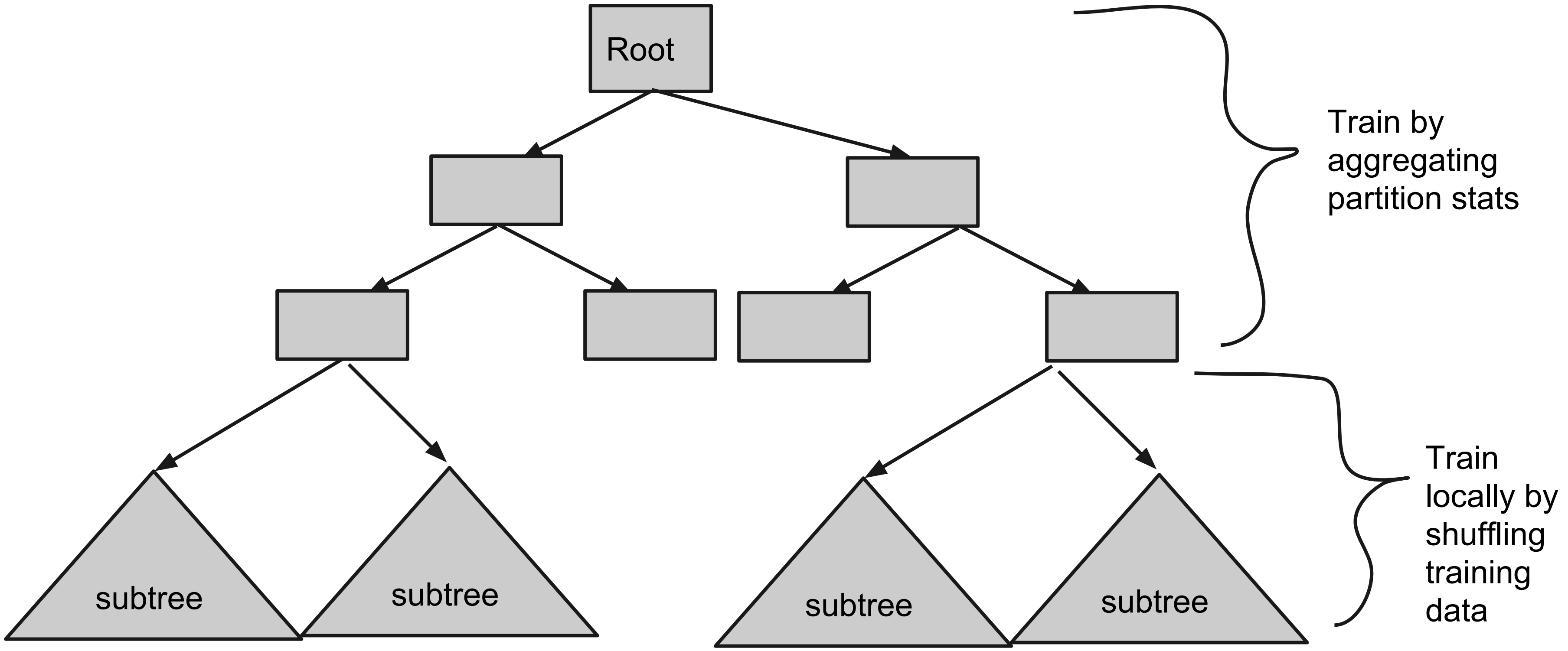
Random Features per Node

- Unlike a single decision tree, we need to randomly select a smaller number of features per node.
- To synchronize among different executors, the driver passes random seeds for different trees to executors. The executors add the node Ids to the seeds and use them to randomly select features for different tree nodes.

Local Sub-tree Training

- Although the first few levels of trees see a lot of training examples, descendant nodes see fewer and fewer training examples.
- Once the number of training samples get small enough, we train small sub-trees locally by shuffling sub-tree training samples to matching executors.

Distributed Sub-tree Training


Root
Train by aggregating partition stats
Train locally by shuffling training data
subtree
subtree
subtree
subtree

The Overall Recipe

1. Train the nodes in a breadth-first manner.
2. Aggregate partition statistics and find optimal splits.
3. If a child of a split has a size smaller than local_threshold, shuffle matching data to an executor and train locally.

Performance factors

1. Time it takes to perform distributed node splits.
2. Time it takes to locally train all the sub-trees.

Performance dependencies

- The performance of partition statistics aggregation is determined by two factors.
  - CPU-bound statistics collections at individual executors. This should scale linearly with the number of executors.
  - The size of the statistics and the message passing/network/shuffle speed. The size is proportional to #_of_nodes * #_of_features *

Performance dependencies

- For information gain, the statistics size is equal to the number of target classes.
- For regression, the statistics size is equal to 3 - sum, square_sum, count.

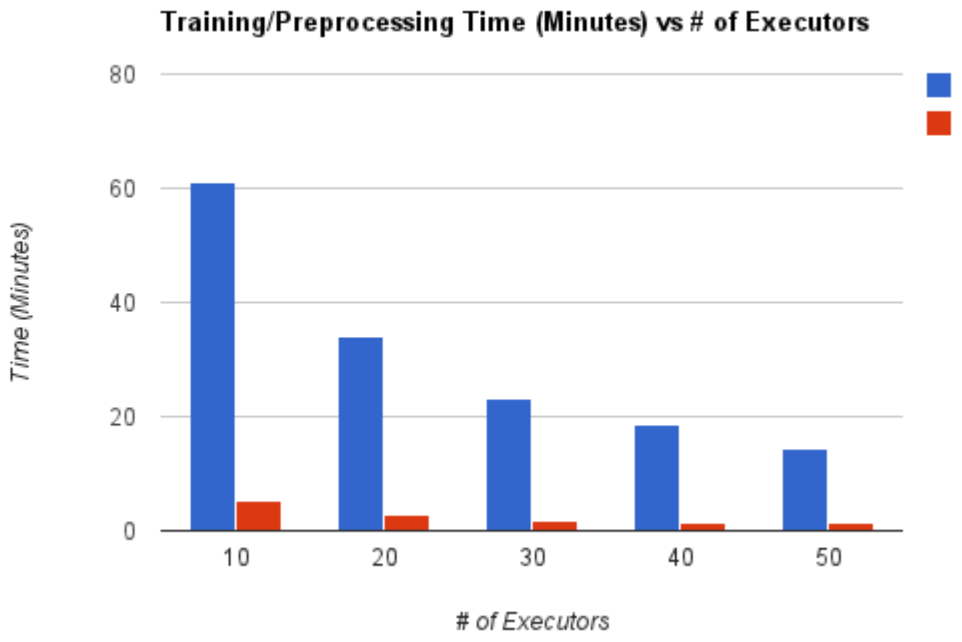
Performance dependencies

- For local sub-tree training, the performance is largely embarassingly parallel - performance almost scales linearly with the number of executors.
- The training data shuffle performance depends on disk-IO and network bandwidth.

Performance scalability

- *MNIST* 8 million samples (784 features, 10 classes).
- 100 Trees
- No limit on tree depth or size.
- # of random features 28 (sqrt(784))
- 100% sampling without replacement.
- 9 machines in AWS, each with 32 cores.
- Measure the time to train vs the number of executors.

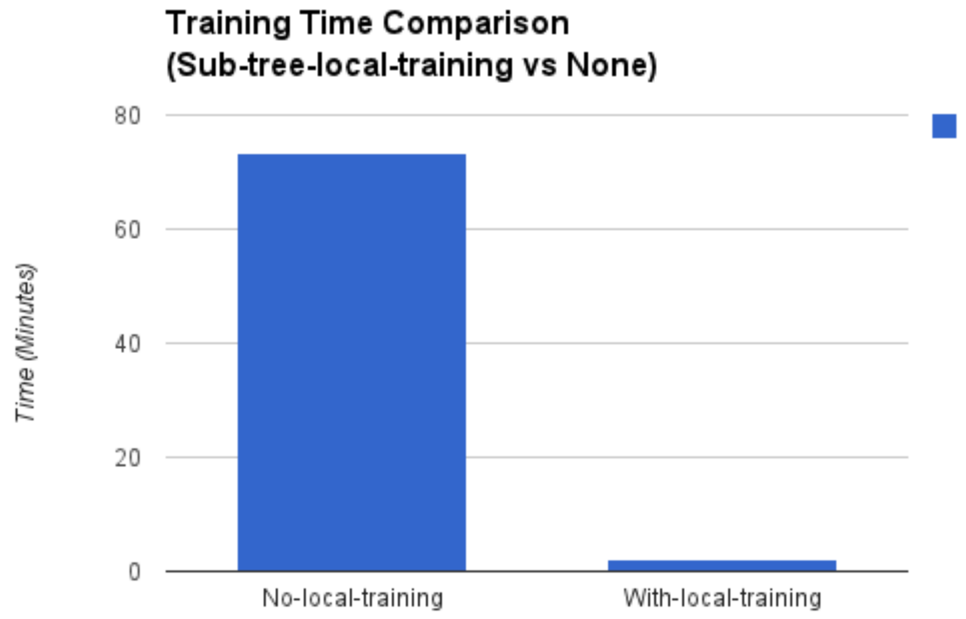
Performance scalability



- Blue : Training Time
- Red : Parse/Preprocess

* The number of machines is fixed at 9. So this doesn't tell the whole story. E.g., we noticed that adding more executors within the same machine will actually slow down the algorithm due to disk IO bottle neck during groupBy/shuffle.

Benefit of local training

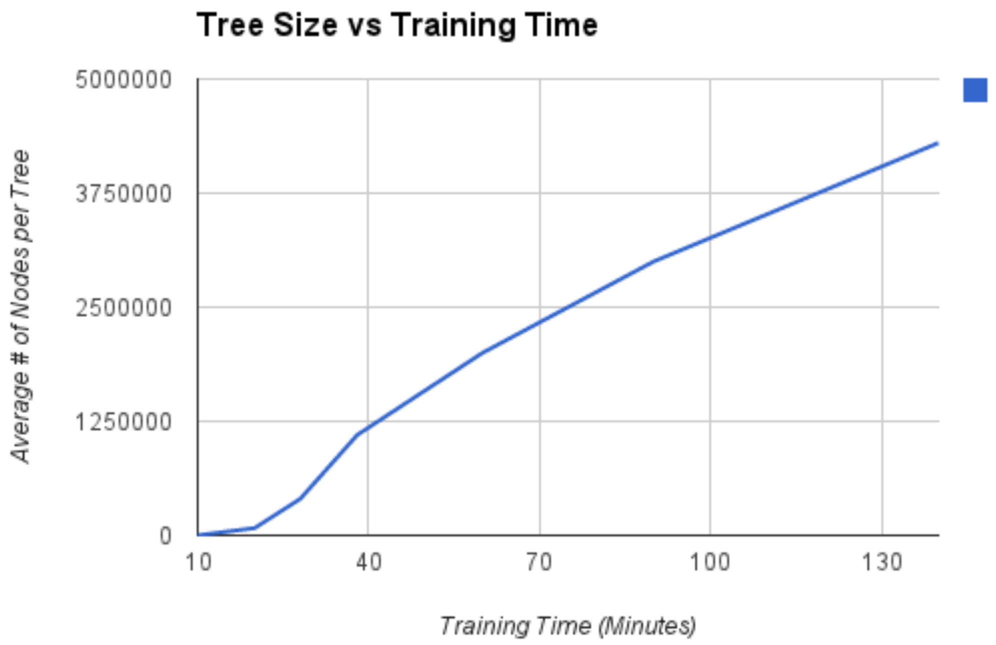


- MNIST 8 million
- 10 trees
- 50 executors
- The local sub-tree training improves performance drastically.

Large scale data testing

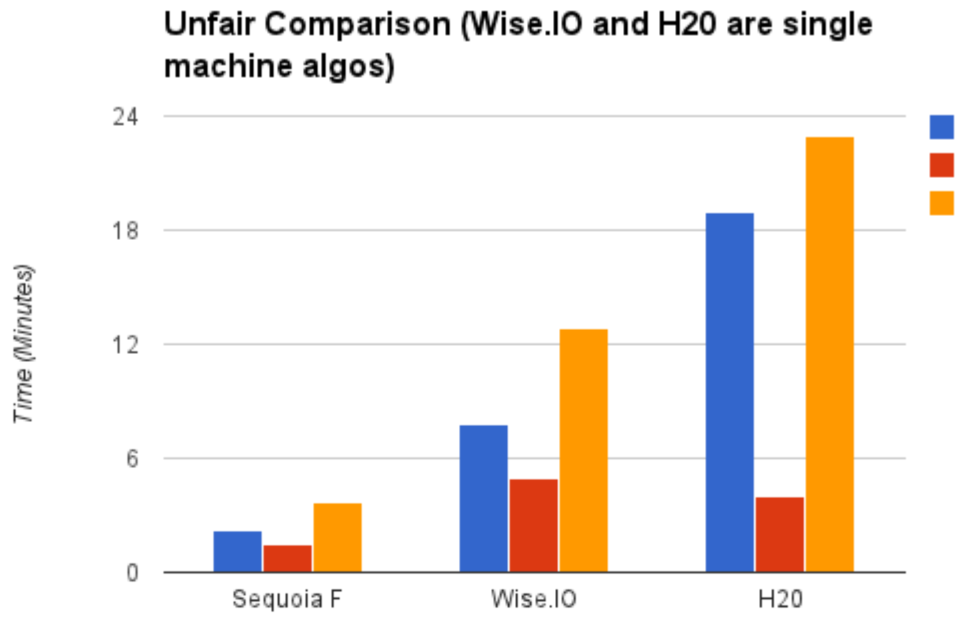
- 200 million samples, 1000 features, 10 classes.
- Affine-transformed *MNIST* data with additional features.
- No tree size limit.
- 100 trees
- # of random features 33
- 100% sampling without replacement.
- 50 Executors

Large scale data performance



- # of nodes per tree goes up linearly with training duration.

Performance comparisons



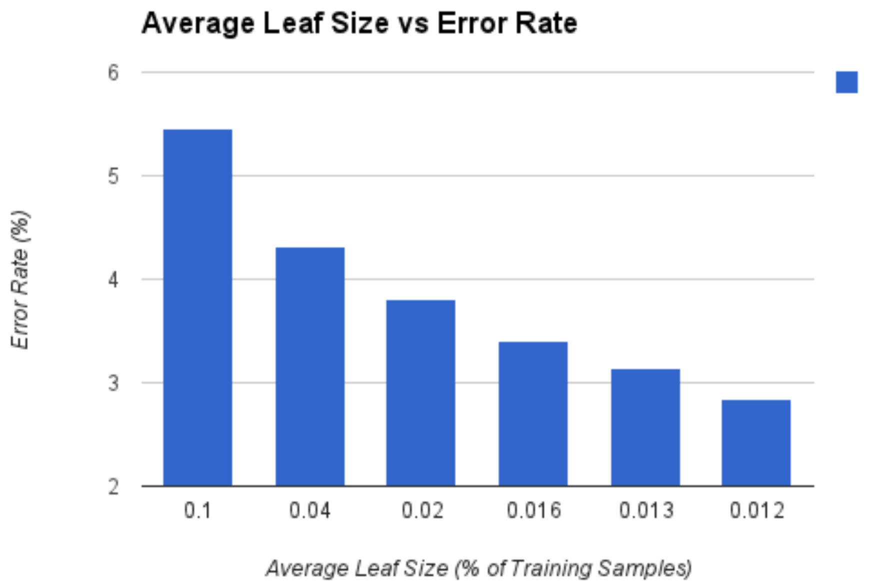
- Blue : Train
- Red : Parse
- Orange : Total
- MNIST 8m
- 10 Trees
- 50 Executors

* Wise.IO, H20 benchmarks are taken from blog entries and are not using the same machine as Sequoia Forest.

Random Forest vs Pruning

- Typically, Random Forest trees are not pruned. The following graphs show how the performance can be affected when leaf sizes get larger.

Accuracy vs Leaf Sizes



- 100 Trees
- MNIST 60000
- Testing data MNIST 10000

Pruning for large scale data ?

- For billions of rows, individual unpruned trees could be too large, even reaching hundreds of millions of nodes per tree.
- To save space and/or training-time, need to investigate whether some pruning could be beneficial from the accuracy-efficiency trade-off point of view.

Shameless Alpine Ad.

Alpine = Spark hadoop Scala

- Alpine is one of the very first companies to receive an official Spark certification.
- We’re looking for brilliant machine-learning/platform/QA engineers.

Conclusion

- Spark enables what was previously impossible - training many fully grown humongous trees on ‘big data’.
- The performance scales well with # of executors.
- Further studies needed to understand accuracy and tree-size trade-offs.