Improving Ensemble of Trees in MLlib

Jianneng Li, Ashkon Soroudi, Zhiyuan Lin

Abstract

We analyze the implementation of decision tree and random forest in MLlib, a machine learning library built on top of Apache Spark, and investigate parallelization as well as algorithmic improvements. We make contributions on two fronts in this paper. First, we analyze the implementation of decision tree and random forest in MLlib, and propose a parallelized version of calculating split bins, or candidate cut-points, for each feature. Second, we discuss the implementation of a tree learning algorithm extra-trees using the existing structure from the random forest implementation, and test both ensembles on multiple datasets. Our results show that the parallelization improvement scales linearly, and that extra-trees perform better than random forest for deeper trees and dense datasets.

1 Introduction

Decision tree is a well-known machine learning algorithm, and has been used successfully to solve a variety of problems including classification and regression [4, 5]. One main reason for its popularity is the model's interpretability: given a decision tree, one can easily identify its prediction on any input by traversing from the root node.

However, despite decision tree's desirable features, it tends to overfit the training data. This means that while the tree may have very low error rates on the data it is trained on, the trained tree's error rate on the test data can be drastically higher. Classic techniques such as cross validation can be applicable in this case to reduce overfitting, but the effect is limited. As a result, variants of decision tree are used instead in practice. One variant of decision tree commonly used is called random forest [1]. A random forest trains an ensemble of decision trees, and for each of them, grows a weaker decision tree. At prediction time, each of the decision trees makes a prediction, and the final decision is chosen through majority vote.

As the digital age advances, we collect more and more data today. In turn, being able to analyze these data and derive insight is more important than ever, and traditional methods of using a single machine to perform the number crunching may no longer be sufficient. In response, a number of parallel frameworks have been developed to process large amounts of data.

For this project, we work with Spark, a general engine designed for large-scale data processing. More specifically, we study the implementations of decision tree and random forest in MLlib, a machine learning library built on top of Spark. We make two main contributions:

- We parallelize MLlib's implementation of decision tree further by calculating the split bins of each feature in parallel rather than serially
- We implement a variant of the random forest algorithm called extra-trees, and perform benchmarks to compare their performances

The remainder of the paper is organized as follows: Section 2 discusses the necessary background in order to understand our accomplishments, including the training process of decision tree and random forest, how Spark performs parallel computation. Section 3 discusses our motivation for working on this project. Section 4 discusses in detail the implementation of decision tree and random forest in MLlib, and points out the changes we made for our contributions. Section 5 performs various benchmarks and analyzes the results. Finally, Section 6 makes closing remarks and offers directions for future research.

2 Background

2.1 Decision Tree

```
1: function TrainDecisionTree(S)
2:     if StoppingCriteriaMet(S) then
3:         if y = 0 for most of ⟨x, y⟩ ∈ S then
4:             return Leaf(0)
5:         else
6:             return Leaf(1)
7:         end if
8:     else
9:         (split, S_0, S_1) ← ChooseBestSplit(S)
10:        return Node(split, TrainDecisionTree(S_0), TrainDecisionTree(S_1))
11:    end if
12: end function
```

Figure 1: High-level pseudocode for training a decision tree on a binary classification dataset.

Given a collection of data points and labels, a decision tree is trained by growing its nodes according to a number of training parameters. Figure 1 shows a high-level algorithm for training a 0-1 binary classification dataset. Recursively, it performs binary partitioning on features, with each bottommost partition having the same label. The algorithm finds the best split using the $\langle$`feature, number of bins`$\rangle$ pair in each node, in order to minimize some impurity function. The term information gain is used to denote the difference between the parent node impurity and the weighted sum of the two child node impurities, where node impurity measures the homogeneity of labels at each node.

Decision tree can stop growing from several stopping criteria. A popular one is to stop when depth of node is equal to some pre-specified max depth. Another possible condition is that none of the split candidates' information gain is greater than some minimum value. The tree also often stops growing when the number of data points in a partition is below some minimum threshold.

2.2 Random Forest

Random forest [1] differs from decision tree in 3 major parts. First, rather than training a single tree, it trains multiple trees, and uses majority ruling during prediction time. Second, when decision the best way to split a node, instead of considering all features, it considers only a random subset of them each time, for example, $\lfloor\sqrt{\texttt{numFeatures}}\rfloor$ features. Third, when training a tree, instead of using all input data, random forest performs bootstrapping, which means it samples with replacement a certain number of points to use for training.

2.3 Spark and MLlib

As mentioned previously, Spark [7] is a general engine for performing parallel computation. It uses a master-worker model, and offers high-level functional API to perform operations on distributed data. Its model of distributed data is called Resilient Replicated Datasets (RDD), which is a read-only collection of objects partitioned across the workers. Each operation is scheduled at the master, performed at the workers, and with results collected back to the master as needed. To speed up computation, Spark tries to cache RDD in memory when possible.

3 Motivation

Our motivation for this project is two-fold. First, the algorithms for training a decision tree or random forest are not inherently parallel. For example, the algorithm for growing decision trees implicitly assumes that all of the training data are in one place, and it takes roughly the same time to access any data point. In a distributed setting, this assumption is not necessarily true. If data is partitioned across a number of machines,

then accessing a remote data point will take significantly longer than accessing a local one. For the algorithm to work well, it needs to be modified to make the nodes effectively communicate with each other.

The second reason for this project is that while researching about tree learning, we came across a paper that describes a variant of random forest called extra-trees [3]. Extra-trees differs random forest in that when splitting a node, instead of choosing the best out of multiple cut-points for each feature in the chosen subset, chooses one cut-point at random. This technique constructs trees even more randomized than those in random forests, and the authors claim that the randomization helps the algorithm to further combat overfitting while being more computationally efficient. We were interested in replicating the paper's results, and learn about the effects of randomization on tree learning.

4 Implementation

4.1 Implementation of Decision Tree and Random Forest in MLlib

```
1: function MLlibTrainDecisionTree(S)
2:     splitBins ← findSplitBins(S)
3:     nodeQueue ← Queue()
4:     nodeQueue.enqueue(Node.emptyNode())
5:     while nodeQueue not empty do
6:         nodes ← dequeueNodes(nodeQueue)
7:         findBestSplits(S, splitBins, nodes, nodeQueue)
8:     end while
9: end function
10:
11: function findBestSplits(S, splitBins, nodes, nodeQueue)
12:     statistics ← collectStatistics(S, splitBins, nodes)
13:     splits ← findBestSplit(S, splitBins, nodes, statistics)
14:     for (node, split) ∈ zip(nodes, splits) do
15:         if StoppingCriteriaMet(node, split) then
16:             makeLeaf(node, split)
17:         else
18:             node.split()
19:             nodeQueue.enqueue(node.leftChild)
20:             nodeQueue.enqueue(node.rightChild)
21:         end if
22:     end for
23: end function
```

Figure 2: High-level pseudocode for the implementation of decision tree training in MLlib.

Figure 2 shows an overview of the algorithm MLlib uses to train a decision tree. Before training, MLlib samples the input data, and puts them to into a certain number of bins. This means that the maximum number of tested cut-points for a feature will be the number of bins for that feature minus one.

To parallelize training, MLlib's implementation of decision tree partitions the input data across its workers. During training, each node that needs to be split is put onto a queue. At each iteration, some number of nodes is pulled from the queue, depending on the available memory. Then, depending on the impurity function, statistics are collected across the partitioned dataset. The statistics for the same node are then shuffled to the same worker, which decides the best splits for the node. Finally, these best splits are sent back to the master, which grows the tree by creating more nodes, adding them into the queue as needed.

Under the cover, a decision tree is treated as a random forest with one tree. MLlib trains multiple trees at once by adding nodes from all of them to the queue.

4.2 Parallelizing Calculation of Feature Split Bins

In the current implementation, we noticed that when computing the split bins during initialization, MLlib uses a single thread on the master to iterate over all features and compute the values. While this method works for small number of features, it is very inefficient for larger numbers. In our optimization, we convert the data structure that will hold the binning information into a RDD, and perform the calculation of feature split bins in parallel using RDD operations. In the end, we aggregate results back to the master to rebuild the data structure.

4.3 Implementing Extra-Trees

In order to implement extra trees, we needed to make a number of changes and additions to the random forest implementation. As we are only testing one random split per feature, extra trees do not need to pre-compute split bins. MLlib's random forest samples the input dataset, and sorts the sample to compute the split bins. In our case, we only need to keep track of the maximum and minimum for each feature. When splitting a node, our extra-trees choose a random split uniformly from the range between the maximum and minimum values of the tested feature. Thus, when selecting the group of nodes to train on for one iteration, in addition to choosing a subset of features for each node, extra-trees will also choose a random cut-point for each of these features.

However, simply using the pre-computed minimum and maximum values will potentially produce meaningless splits. For example, if the parent had previously split on (feature 5, ≤ 1.5), then the randomly chosen split (feature 5, ≤ 2.0) for the child with feature 5 less than 1.5 would not result in any information gain. Therefore, each node must also store the updated maximum and minimum values for features that its parents had split on. We have each node keeping a hash table to store the updated minimum and maximum values, where the key is the feature number, and the value is the updated value. When choosing a random split for each feature of a node, we first check the hash table for updated values of the tested feature, before using the pre-computed minimum and maximum values. The extra memory required for this hash table is small, as the number of values stored is equal to the depth of the node.

For parallelization, we have each worker compute statistics for each split, similar to the implementation of random forests. The key difference is that, instead of computing statistics for each bin of every feature, we only test on one split per feature, so we only need to keep track of 2 statistics per feature: one for data to the left of the split threshold, and one for data to the right. Random forests are typically run with the maximum bins per feature set to 32. Therefore, extra-trees stores less data for statistics, with a maximum reduction of 16 times less data when dealing with very dense data with a wide range of values per feature. In addition to memory, this also reduces the amount communication between workers, as these statistics are aggregated and re-broadcast later on.

The last change we had to make for the extra-trees algorithm is to make sure that when choosing data points to train a tree, we do not bootstrap, and instead use all of the available input.

5 Evaluation

5.1 Setup

For all of our evaluations, we ran our benchmarks on EC2. There is one Spark master node running on c3.xlarge, which has 4 cores and 7.5 GB of RAM, and 4 Spark worker nodes running r3.large, each having 2 cores and 15 GB of RAM. These machines are in the same region, availability zone, and placement group, so they have fast network connections to each other.

Aside from the ones shown below, our benchmarks use the same parameters for everything else. This includes the impurity function (entropy), maximum number of bins per feature (32), minimum information gain before stop splitting (0), and minimum number of data points before stop splitting (1).

5.2 Calculation of Split Bins

We tested the parallelized version of finding split bins against the serial version with several datasets of varying feature counts. The serial implementation ran on the master node, while the parallel implementation ran on up to 4 worker nodes. The results are shown in Figure 3. When we use fewer than approximately 10,000 features,

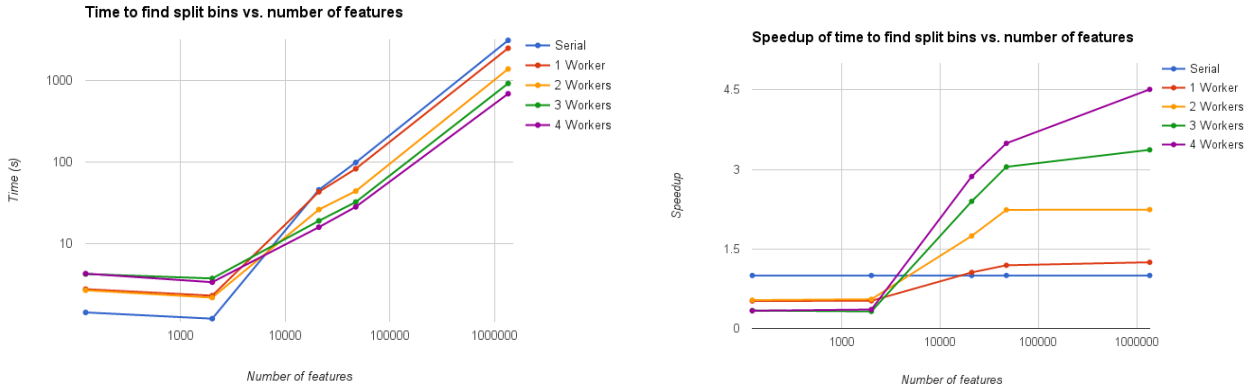

Figure 3: Performance comparison between the serial and parallel implementations of finding feature split bins.

the serial version performs better, due to the overhead of parallelization. As we increase the number of features, the parallel implementation gains more and more of an advantage.

Our parallelization improvement also shows clearer strong scaling properties as the number of features increases: at over 1 million features, speedups of 1.2, 2.2, 3.4, and 4.5 are observed for 1, 2, 3, and 4 workers respectively. The numbers make sense, because the calculation for each feature is independent, which means no coordination is required between workers. The reason that the speedups are greater than the number of workers is that within each worker, Spark can use multiple threads to further parallelize processing.

5.3 Extra-Trees

For these benchmarks, we used datasets from the LibSVM data library [2].

5.3.1 Dataset #1: MNIST Digits

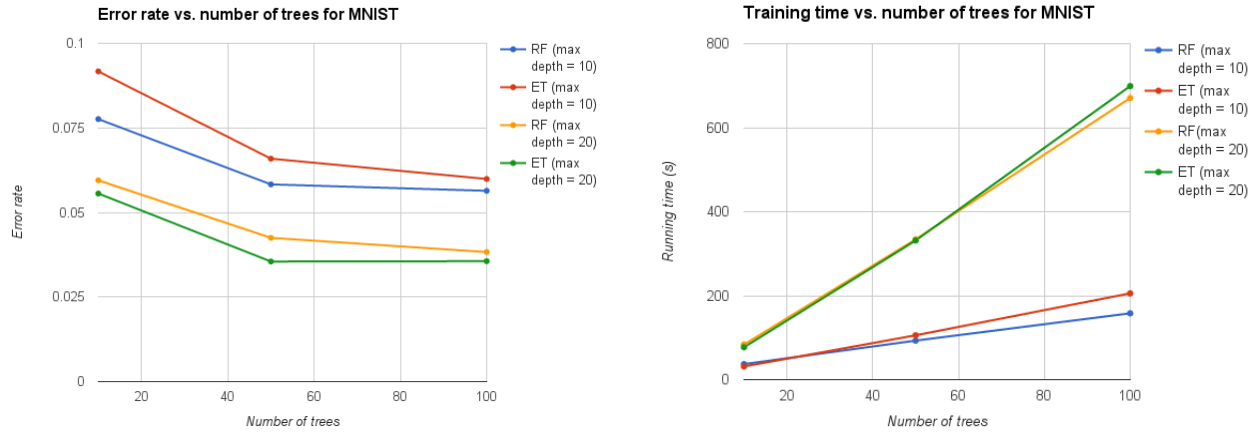

Figure 4: Performance of random forest (RF) and extra-trees (ET) on the MNIST dataset.

This dataset consists of images from the MNIST digit dataset, in which each data point is represented with 28x28 = 784 features. Each feature is an integer value, from 0 through 255, representing the intensity of the associated pixel. This is a classification problem 10 classes. Our results are in Figure 4.

When limiting the max depth to 10, extra-trees are less accurate and slightly slower than random forests, likely caused by the extra randomization. When we increase the max depth to 20, extra-trees take around the

same time to train as random forests, but are more accurate. The improved accuracy can be attributed to the randomization's help against overfitting. However, because extra-trees test on fewer, and random splits every node, the average information gain per node is lower than that of random forests. As a result, extra-trees often have more nodes before they converge. From these results, we can deduce that extra-trees benefit more from deeper trees than do random forests.

5.3.2 Dataset #2: GISETTE

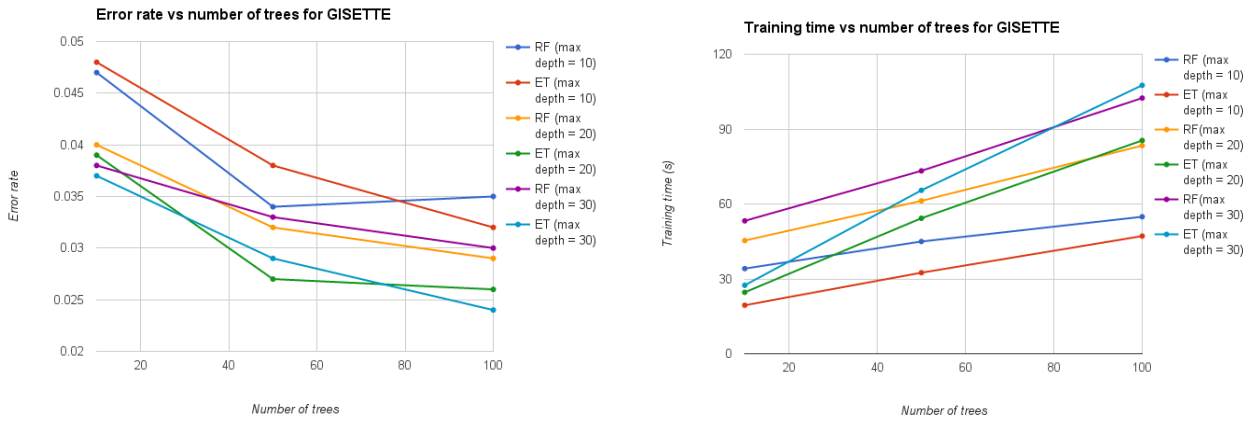

Figure 5: Performance of random forest (RF) and extra-trees (ET) on the GISETTE dataset.

This dataset is from the NIPS 2003 feature selection challenge. It consists of digits from the MNIST data set with the goal of distinguishing 4's from 9's, making it a classification problem with 2 classes. Higher order features were created from samples of pixels, and some "distractor" features were added that have no predictive power. Finally, the features values are scaled to be between -1 and 1. The featurization makes this dataset particularly suited towards decision trees, which will ignore these bad features due to their low information gain. There are a total of 5000 features for each point in this set. Our results are in Figure 5.

In this dataset, extra-trees is the clear winner. When depth is limited to 10, performance is roughly equivalent in terms of error rate, with extra-trees having an advantage in speed. For depths of 20 and 30, extra-trees performs better in accuracy at all tree-counts. This is more evidence that extra-trees benefit more from deeper trees. In addition, extra-trees may benefit more from larger feature-sets, as it provides more opportunity to find a good split at each node.

5.3.3 Dataset #3: IJCNN1

This dataset is from the IJCNN 2001 Neural Network Competition. This is a classification problem with 2 classes. The original 5 features are transformed into 22 features using the winner's transformation. This dataset has more sparse data compared to the others, and also fewer features. Our results are in Figure 6.

In this dataset, random forests are clearly more accurate and have faster training times at all depth levels. This is likely because of the few and sparse features. When there are fewer features available, both random forest and extra-trees suffer from choosing a square root subset of features during splits. Additionally, because the feature values are sparse, there is no noticeable time gain of testing one cut-point over testing all possible cut-points per split.

5.3.4 Dataset #4: Year Prediction MSD

This data is from the Million Song Dataset. Each data point contains 90 different features related to the song, and the goal is to predict the year the song is from, i.e. a regression problem. Since all data points have values for all features, this is a dense dataset. Because both tree models take longer to train, we only ran them with a subset of our other tests. The results are in Figures 7 and 8.

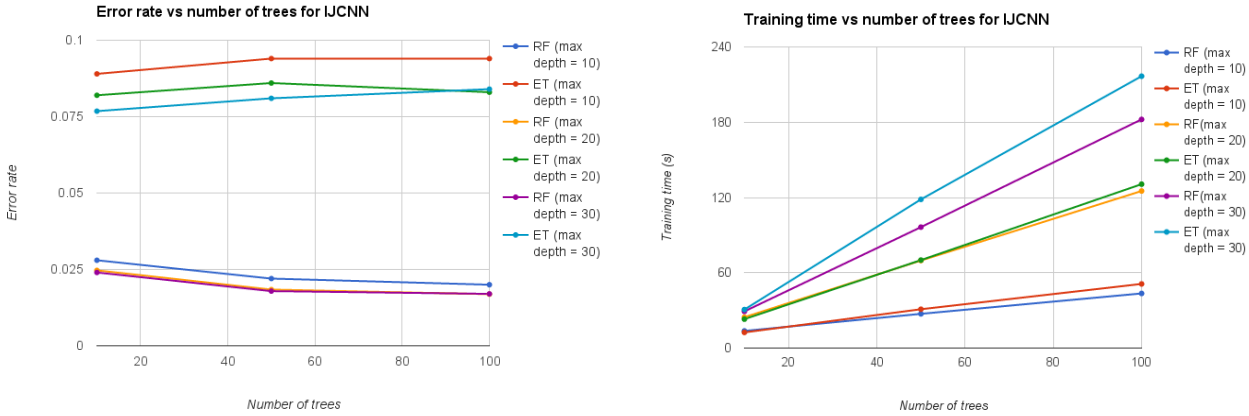

Figure 6: Performance of random forest (RF) and extra-trees (ET) on the IJCNN11 dataset.

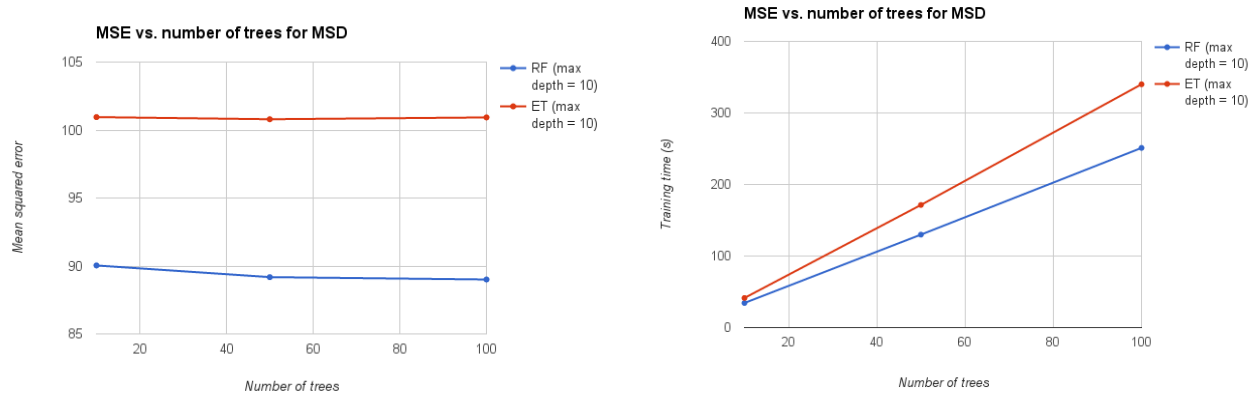

Figure 7: Performance of random forest (RF) and extra-trees (ET) on the YearPredictionMSD dataset for max depth of 10.

For this dataset, random forests performs better than extra-trees and takes less time to train when the max depth is 10. However, when we increase the max depth to 20, extra-trees have about the same mean squared error as random forest, while taking many times shorter to train. Both models have significantly more nodes per tree than when training other datasets, so the speed difference can be attributed to the fact that extra-trees are able to train more nodes per iteration, since they consume less memory.

5.3.5 Scaling of Training Time

Aside from comparing accuracies and training times of random forest and extra-trees, we also looked at the scaling characteristics of extra-trees' training time. As seen from Figure 9, the training times for MSD show good strong scaling, while the training times for MNIST do not. This discrepency is likely due to the size of the two datasets: MNIST has 60,000 data points, while MSD has 463,715. When training smaller datasets, it is more difficult for the benefits of parallelization to overcome the overhead.

5.3.6 Discussion

From our tests on different datasets, we can make some observations about the characteristics of extra-trees. First, extra-trees perform better when allowed to train with deeper trees. While training deeper trees can often

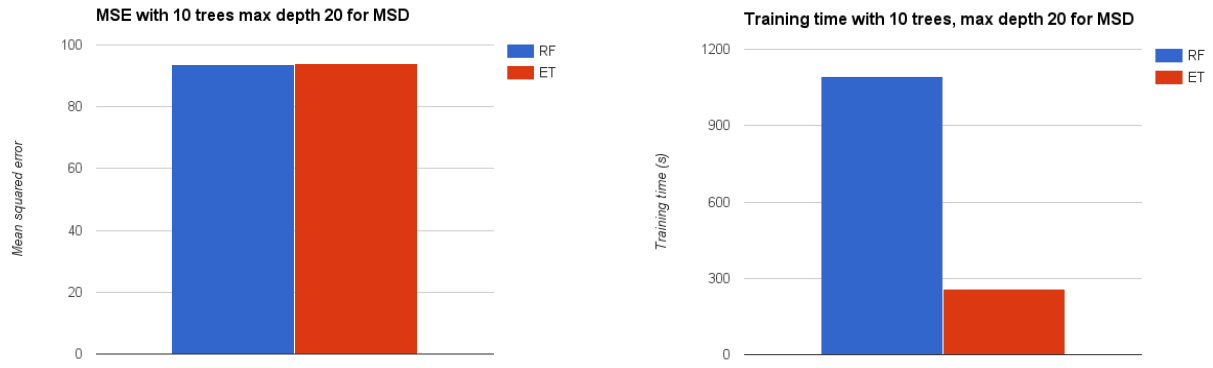

Figure 8: Performance of random forest (RF) and extra-trees (ET) on the Million Song Dataset (MSD) dataset for 10 trees and max depth of 20.

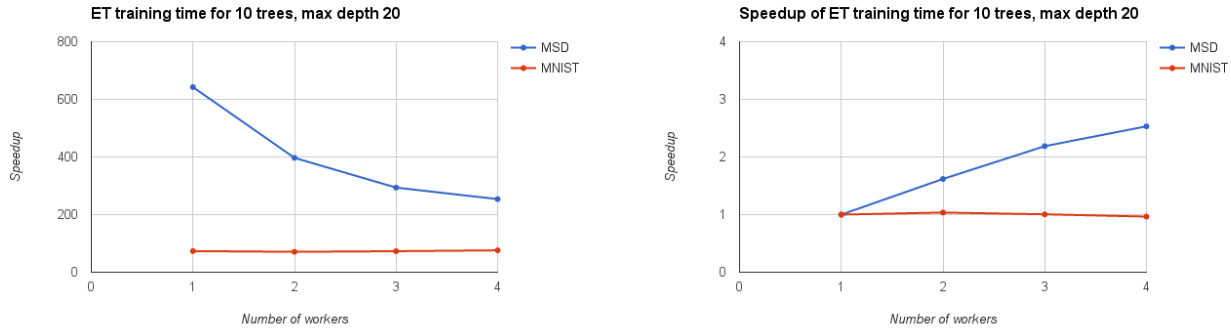

Figure 9: Scaling extra-trees (ET) training time for Million Song Dataset (MSD) and MNIST dataset.

lead to overfitting and increased test error, this effect is less pronounced for extra-trees due to the randomness. It is likely that, because of lower information gain per node on average, extra-trees need deeper trees to be able to be as accurate as random forests.

Another effect we see is that extra-trees perform better when we have larger datasets with more dense features. By testing on more features per split, the chances for extra-trees to find a very good split for a node is increased. When data is more dense, there is also often a wider range of values for each feature. This results in more cut-points per feature for random forests, giving extra-trees a larger advantage in time needed to split a node.

Moreover, extra-trees require less memory use than random forests, especially when dealing with features with a wide range of values and thus more potential cut-points. When dealing with more nodes on deeper trees, there may not be enough memory to train all nodes in one level in a single iteration. Extra-trees gain an advantage here by being able to process more nodes per iteration, resulting in a faster training time.

In terms of scaling, when the dataset is sufficiently large, extra-trees trains proportionally faster as more workers are added. For smaller datsets, parallelization is not as helpful. In fact, modern computing hardware should be enough to train small datasets even serially at a reasonable time.

6 Conclusion and Future Work

In this paper, we described two contributions we made on Spark MLlib's tree learning algorithms. One accomplishment is optimizing the calculation of split bins on features by performing the operation in parallel. With

more features, the speedup becomes more apparent. Using 4 workers, we are able to achieve up to more than 4x the performance of the original serial implementation as the number of features increases to more than 1 million.

Our second contribution dealt with implementing the extra-trees algorithm in MLlib. Unlike random forests, extra-trees choose a random split for each feature, and do not bootstrap data. We found that, in terms of performance, extra-trees have better accuracies for deeper trees and datasets with larger number of features. In terms of training time, they run faster for dense data with a large range of values for each feature.

There are a number of possible directions for future work. For example, aside from changing the maximum depth, we used the default parameters for stopping condition, which are when information gain is 0, or the number of data points on a node is 1. It would be interesting to compare how the performances of random forest and extra-trees change as we tweak the stopping conditions. Also, since we only focused on datasets with continuous feature values, it is worth looking into whether similar results can be obtained when feature values are categorical.

Lastly, due to resource constraints, we only used a Spark cluster with up to 4 workers. Since Spark is designed to run on hundreds of machines handling terabytes, if not petabytes of data [6], it would be good to run our experiments on many more workers, and observe the scaling properties.

References

[1] L. Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[2] R.-E. Fan, C.-C. Chang, and C.-J. Lin. Libsvm data: Classification, regression, and multi-label. `http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/`, 2011. [Online; accessed 01-May-2015].

[3] P. Geurts, D. Ernst, and L. Wehenkel. Extremely randomized trees. *Machine learning*, 63(1):3–42, 2006.

[4] A. M. Prasad, L. R. Iverson, and A. Liaw. Newer classification and regression tree techniques: bagging and random forests for ecological prediction. *Ecosystems*, 9(2):181–199, 2006.

[5] S. R. Safavian and D. Landgrebe. A survey of decision tree classifier methodology. 1990.

[6] R. Xin. Spark the fastest open source engine for sorting a petabyte. `https://databricks.com/blog/2014/10/10/spark-petabyte-sort.html`, 2014. [Online; accessed 05-May-2015].

[7] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.