Distributed Random Forest: Final Report

Sumeet Mishra
sumish@iu.edu
Indiana University Bloomington

Srinithish K
skandag@iu.edu
Indiana University Bloomington

1 INTRODUCTION

Cluster computing becomes popular when the cost of commodity PC hardware and network equipment dropped. However, there are limitations to cluster computing. Network communication becomes a severe bottleneck, and there always exists the possibility of node failures within the cluster. The Distributed Random Forests algorithm specifically targets implementation in cluster environments and attempts to accommodate the inherent limitations and concerns of using cluster hardware by presenting an algorithm with sparse communication and fault tolerance. It is a powerful classification and regression tool. When given a set of data, it generates a forest of classification or regression trees, rather than a single classification or regression tree. Each of these trees is a weak learner built on a subset of rows and columns. More trees will reduce the variance. Both classification and regression take the average prediction over all of their trees to make a final prediction, whether predicting for a class or numeric value.

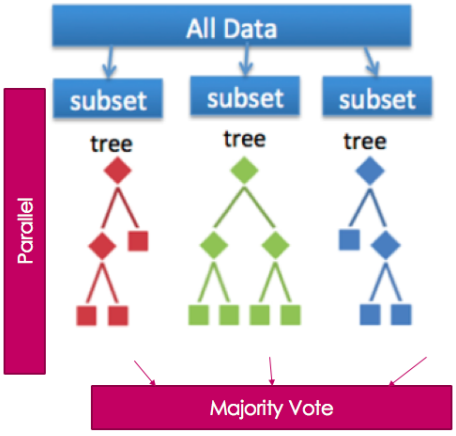

Figure 1: Distributed Random Forest

1.1 Apache Spark

Apache Spark has been one of the most widely used methods of performing large scale batch processing or stream data processing on distributed systems.

This open source analytics engine stands out for its ability to process large volumes of data significantly faster than MapReduce because data is persisted in-memory on Sparks own processing framework. It is highly flexible and scalable with support in many programming languages like Python, Scala, Java, SQL, R, etc and can run on standalone cluster mode, on EC2, on Hadoop YARN, on Mesos, or on Kubernetes. Also, Access data in HDFS, Alluxio, Apache Cassandra, Apache HBase, Apache Hive, and hundreds of other data sources. Due to these extensive support options along with built-in libraries for running Machine Learning programs it has become an important platform to run algorithms on distributed systems.

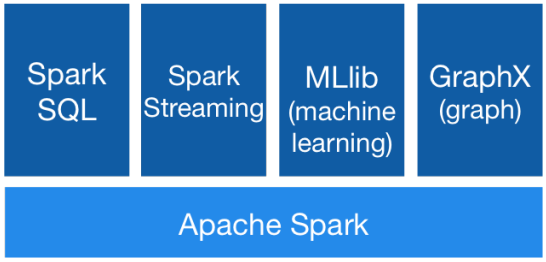

Figure 2: Apache Spark Framework

1.2 Tensor Flow

TensorFlow is an open source software library for high-performance numerical computation. Its flexible architecture allows easy deployment of computation across a variety of platforms (CPUs, GPUs, TPUs), and from desktops to clusters of servers to mobile and edge devices. It creates the whole network with placeholders. Placeholders enable us to separate the network configuration phase and the evaluation phase. It actually helps to scale up the network without any modifications being made.

2 RELATED WORK

We found some related work as follows-
Tao et al. [6] and Lin et al. [7] introduced some classification algorithms for high-dimensional data to address the issue of dimension-reduction. Strobl [8] and Bernard [9] studied the variable importance measures of random forest and proposed some improved models for it. Taghi et al. [10] compared the boosting and bagging techniques and proposed an algorithm for noisy and imbalanced data. Yu et al. [11] and Biau [12] focused on RF for high-dimensional and noisy data and applied RF in many applications such as multi-class action detection and facial feature detection, and achieved a good effort. Basilico et al. [13] proposed a COMET algorithm based on MapReduce, in which multiple RF ensembles are built on distributed blocks of data. A Parallel Random Forest Algorithm for Big Data in a Spark Cloud Computing Environment by Jianguo Chen et al[14]. Random Forests for Big Data by RobinGenuera et al[15] disscusess how out-of-bag error is addressed in parallel adaptations

of random forests.

3 METHOD

3.1 Data

We used Airlines data for the year 2008. Since the data does not have a specific target variable, we decided to predict if the flight got delayed at its arrival. Any flight that got delayed by 15 minutes or greater is considered delayed. A variable 'isDelayed' was created to convert the problem into classification. Variables 'DepTime', 'CRSDepTime','ArrTime', 'CRSArrTime' all had a time stamp in HH: MM format from which we extracted the time in minutes. The variables 'Dest'(Destination) and 'Flight tail number' was dropped as they are unique and the distance traveled was already captured in another variable. Variables 'DepTimeInMins' was also dropped as we were predicting if the flight got delayed and hence we wouldn't know this variable in the real scenario.
After creating the variable IsDelayed we have the following distribution,

Not Delayed	5541790
Delayed	1313239

The total data set has approximately 7 Million rows.

CRSElapsedTime	AirTime	ArrDelay	DepDelay	Distance	TaxiIn	TaxiOut	DepTimeInMins	CRSDepTimeInMins	ArrTimeInMins	CRSArrTimeInMins	IsDelayed
150.0	116.0	-14.0	8.0	810	4.0	8.0	1203.0	1195	1331.0	1345	0
145.0	113.0	2.0	19.0	810	5.0	10.0	474.0	455	602.0	600	0
90.0	76.0	14.0	8.0	515	3.0	17.0	388.0	380	484.0	470	0
90.0	78.0	-6.0	-4.0	515	3.0	7.0	566.0	570	654.0	660	0
90.0	77.0	34.0	34.0	515	3.0	10.0	1109.0	1075	1199.0	1165	1

Figure 3: Few rows of the data

3.2 Spark

3.2.1 *Installation:*

Spark was installed on top of Hadoop's YARN cluster manager. Pyspark an interface to Spark's functions using Python was used to run programs. Configured the maximum data that could be returned to the driver as 6GB from the default 1024 MB.

3.2.2 *Decision Tree Implementation:*

We implemented a Decision tree from scratch as we wanted greater control over the parameters and functions like getting the best splitting criteria over a variable. Of-the-shelf implementation like Sci-kit Learn, do not give this freedom or its too hard to tweak their functions due to dependencies. Hence we decided to write our own decision tree implementation. We have built a decision tree for classification problems. Randomization of the decision tree in a random forest can be achieved by subsampling the data points and considering a subset of the variables or features for a split in a tree.

3.2.3 *Task Parallel approach:*

The most interesting part of Random forests is the parallelization of the growth of the trees. Since its an ensemble technique once you have the data at every executor, the task of building trees in parallel can happen without much interaction between the executors as the splitting and growing of the tree is only related to that tree and independent of the other trees. Exploiting this feature of the random forest we implemented the parallelization of the building of each decision tree in the forest. Randomization was achieved by subsampling (bagging) of the data points and storing their corresponding indices in a Resilient Distributed Datasets (RDDs). Note that we do not store the subsamples themselves, but only the indices thereby reducing memory footprint. We now get the executors to build decision trees in parallel and get their vote on the test data and store them in an RDD. We now aggregate these votes get the majority by folding the RDD. To reduce the traffic of the data transferred between and to the executors the input data matrix was broadcasted such that the variable is available for all the executors without having to be shipped at every task. Hence building a Parallel Random Forest in Task parallel approach.

3.2.4 *Data Parallel approach:*

We attempted optimization using Data Parallel approach where we would be vertically partitioning the data on each feature and save it in a Resilient Distributed Datasets (RDD) and hence the best split computation can occur close to the data. However, since our data was not so huge and could be fit in single node this approach though included did not actually increase our performance.

3.2.5 *Scaling to multiple Nodes:*

We scaled spark to multiple nodes and experimented with changing several parameters like numExecutors, numCoresPerExecuter

3.3 Tensorflow

3.3.1 *Installation*

. We installed tensorflow inside anaconda environment to better maintain any package dependencies. We installed the CPU version of the tensorflow as Juliet nodes do not have a GPU. We installed Horovod which is an opensource platform by Uber for distributed training.

3.3.2 *Decision Tree implementation*

. We wrapped the decision tree class with the tensorflow pyfunc wrapper so that the tree's decisions were returned as tensors rather than numpy arrays. The sampling indices are broadcasted instead of the data itself to reduce the cross-communication overhead when running parallelly.

3.3.3 *Task parallelism and scaling*

. We used Horovod an MPI based framework for scaling of the training two multiple nodes. We used allgather to collect the results from multiple nodes and then take a majority of the vote. AllGather ensures all process join and synchronize in the end. Horovod starts multiple processes and communicates through MPI. We first generate random indices and then broadcast this set of indices to all the process and only rank 0 processes broadcasts.

4 EXPERIMENTS WITH SPARK

All the experiments are conducted on a single node. For each experiment, we used 10 decision trees with maximum depth allowed in each tree as 5.

4.1 Sequential approach

We implemented the random forest without parallelizing the tree growth. This serves as a benchmark for our experiments. Trees were grown in sequence on a single core.

F1Score	0.8115699
Time for computations(s)	886.382

4.2 Parallel approach

Implemented the tree growth in parallel by allowing the spark to exploit all the cores in the node. The Juliet node has 48 cores in each node.

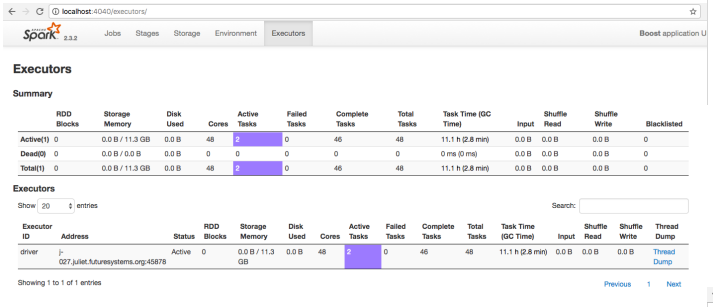

Figure 4: Screenshot of job running on spark framework

F1Score	0.811978
Time for computations(s)	198.502

4.3 Parallel with modified splitting criteria

As an attempt to improve the accuracy, we changed the number of split points to be tried while calculating the best possible split. Initially for computational simplicity split thresholds were tried only at the median of the data present at each split. But to improve the accuracy we allowed the tree to be split at 3 quantiles of the variable. Namely at 25%, 50% and 75%. We improved the accuracy by 3.5% but at a slightly higher computation time about 30 seconds higher.

F1Score	0.83897
Time for computations(s)	232.465

4.4 Parallel with Multiple nodes

We used the nodes that were provided to us that is j-027 and j-026 and ran the spark job.

We observed a bit of communication overhead between two nodes initially while the computing time for the job is less but since the communication overhead is a constant we observed the optimization achieved when the computing time of the job increased.

Below is an example of how the job looks like in a Hadoop cluster.

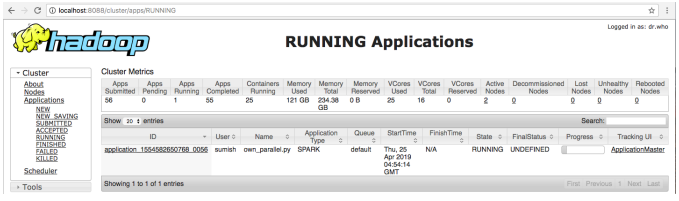

Figure 5: Screenshot of job running on Hadoop cluster

We can check the job details of the running job on Hadoop cluster.

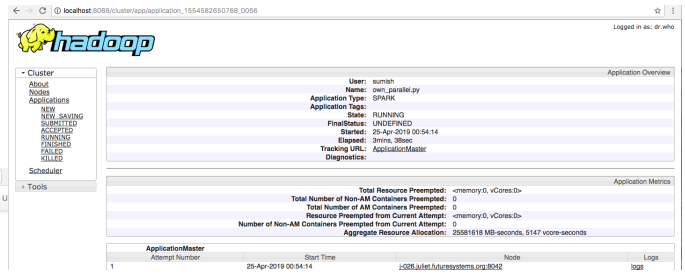

Figure 6: Details of the job running on hadoop cluster

We can also check whether the job is running on both the nodes.

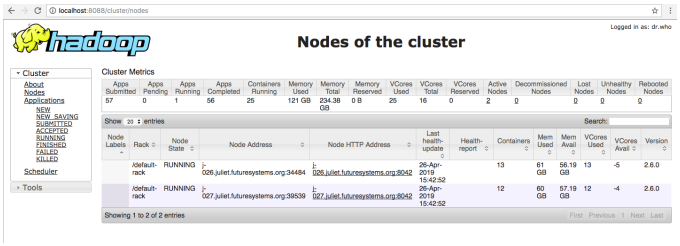

Figure 7: job running on two nodes hadoop cluster

4.5 Challenges

We faced many challenges while scaling into multiple nodes and running spark job on them. The major challenges were,

- Running out of memory for temp folder. Since our job requires more memory than allocated to the temp folder the job failed while running. We took care of this issue by moving our temp folder to a different drive with more memory.
- Executors failed due to memory shortage. Initially, our executors were given a default memory of 1GB, but our executors need more than that for running successfully. We periodically tested and finally increased the memory of all the executors to 5GB as the memory required by the biggest executor was 4GB.
- We faced issued while our random forest algorithm did not find the decision tree class which we wrote in a separate script. We merged the decision tree class to our random forest algorithm. So we are using 2 scripts for single and two node implementation named **sparkSingleNodeParallel.py** and **sparkMultipleNodeParallel.py** respectively where the **sparkSingleNodeParallel.py** is the combination of **sparkMultipleNodeParallel.py** and **decisionTree.py**
- We also faced issues while trying to build and access the User Interfaces like spark and Hadoop which we solved via tunneling into Juliet node from our local system.

4.6 Observations

We have taken almost 50 observations and analyzed different aspects of performance related to accuracy and time took for the job to complete.

We observed that with the increase in the number of trees the time also increases to complete the job.

Nodes	Executors	Cores_per_Executor	Total_Cores	Trees	Accuracy	Time_Taken
1	48	1	48	15	0.83969416	274.128288
1	48	1	48	30	0.83720484	351.7916203
1	48	1	48	40	0.83897535	455.2651787
1	48	1	48	50	0.8390462	705.7197242
1	48	1	48	70	0.83969416	785.5331531
1	48	1	48	100	0.83969416	1390.428936
2	24	4	96	20	0.83897257	386.7260301
2	24	4	96	40	0.83969416	440.6733391
2	24	4	96	60	0.83904343	509.2158473
2	24	4	96	80	0.8390462	610.1735959
2	24	4	96	100	0.83897535	827.9424903

Figure 8: Spark:Observation collected for Time,Trees and Accuracy

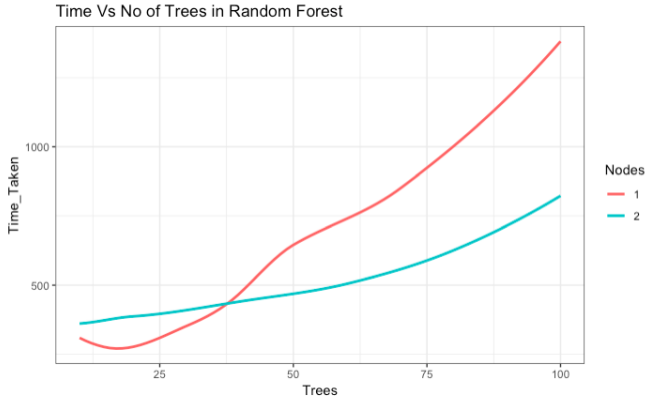

Figure 9: Spark:Time versus Number of Trees

The graph in fig 9 depicts the performance comparison of single and two nodes implementation of random forest. We can observe that after around 32 trees the time taken by a single node to complete the job exceeds the time taken by two nodes. We can assume that the initial time taken by two nodes is high because of the communication overhead of the two nodes. But as it is constant time, therefore the time taken by two nodes decreases when we increase the number of trees.

Now we can observe from the plot in fig 10 that is plotted between Accuracy and number of Trees that even if we increase the number of trees the accuracy does not vary pretty much. We also observed a dip and we can assume that it is happening because of randomization.

We have taken our observation by taking the total cores as constant and varying the number of executors for a single node to check the effect on the computation time as depicted by figure 11. We observed that by increasing the number of executors the computation time also increases.

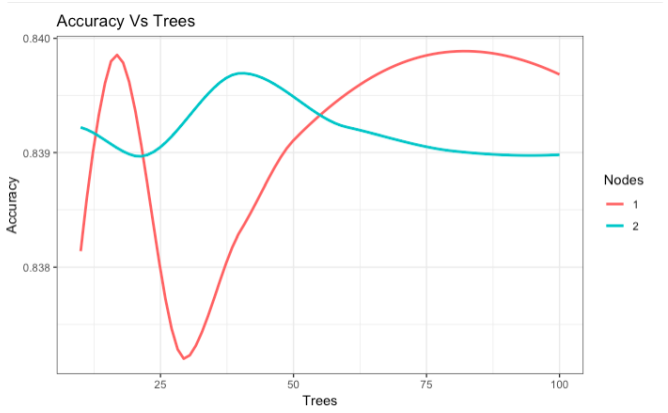

Figure 10: Spark : Accuracy versus Number of Trees

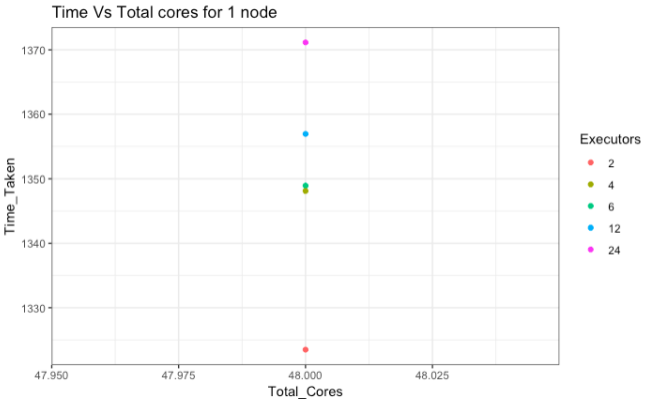

Figure 11: Spark: Time versus Number cores and Executors for single node

Nodes	Executors	Cores_per_Executor	Total_Cores	Trees	Accuracy	Time_Taken
1	24	2	48	10	0.83969416	1371.15842
1	12	4	48	10	0.83897535	1356.947258
1	6	8	48	10	0.83969416	1348.933374
1	4	12	48	10	0.83897535	1348.112859
1	2	24	48	10	0.8390462	1323.483079
2	24	2	48	10	0.83897535	293.274262
2	36	2	72	10	0.83969416	358.5712302
2	20	4	80	10	0.83897535	344.5035419
2	40	2	80	10	0.8382859	367.1295996
2	48	2	96	10	0.83969416	386.4629402
2	50	2	100	10	0.83969416	413.1738794

Figure 12: Spark: Observation collected for Time, Executors,Total Cores and Accuracy

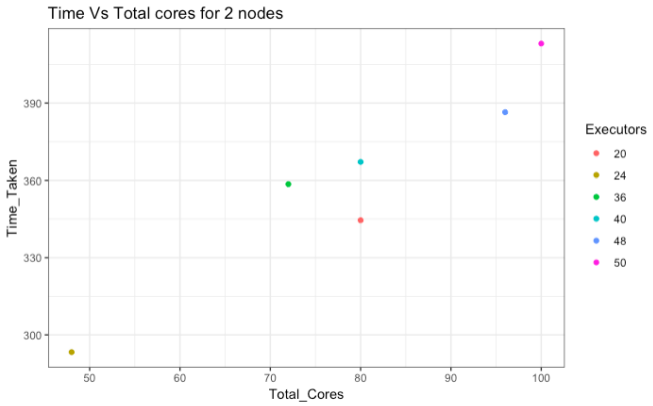

Figure 13: Spark: Time versus Number of cores and Executors for two nodes

For two nodes also we observed by increasing number of executors the computation time increases. Here we varied the total cores along with the number of executors.

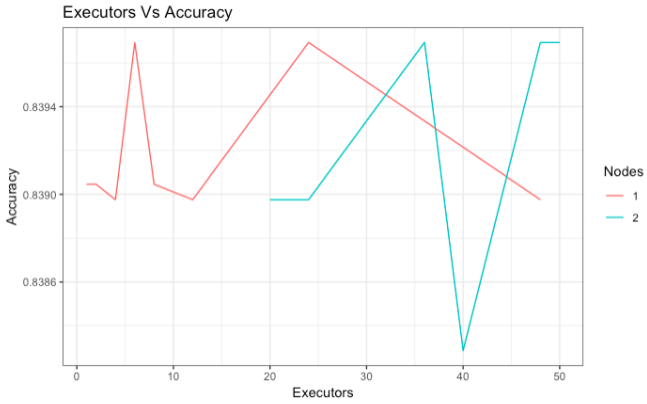

Figure 14: Spark: Executor versus Accuracy

We can observe from the above plot that the accuracy has a random effect on the number of executors. But the interval is very less.

5 EXPERIMENTS WITH TENSORFLOW

5.1 Sequential approach

We implemented a function in tensorflow wrapping our default DecisionTree.fit() function. Which when executed in a loop with 10 trees gave us the following result

F1Score	0.8215699
Time for computations(min)	23.8

5.2 Parallel Approach: Single node

We implemented a parallel approach in tensorflow using MPI based wrapper Horovod. Horovod starts several processes with the same script. First, we generate set of random indices based on the number of trees required to say we had 100 rows and 10 trees were required the indices from 0 to 99 would be put in 10 different sets randomly. Each set containing 2/3 of total rows sampled with replacement. We then broadcast this set of indices to all the process so that the indices set used is the same across all the processes. The indices are broadcasted from the parent (rank 0) process to all other processes. Horovod assigns a rank to each process and size is the total number of parallel processes started. In each of the process, a single tree is built. The data on which a tree is built is based on the rank of the process

index = indices[rank]

Each process both fits and predicts on the test set. The returned predictions for the test set from each of the process (tree) are collected using MPI AllGather. AllGather ensures all process join and synchronize in the end. On the gathered results we take a mode at each row denoting the majority of predictions for that particular example. Hence the final predictions are made.

5.3 Parallel Approach : Multiple nodes

The parallel approach with multiple nodes in Horovod/Tensorflow is rather simple as we just need to add the permission for the 2 nodes used and the rest of the code needs not much change as except for the command to run the script. We achieved a significant gain in performance when using two nodes. However, the overhead threshold did appear for 4 trees when 2 each were split amongst 2 nodes.

5.4 Observations

Using the above approach we got the following results for the single and double node (distributed training). The computational time is in seconds

numTrees	numNodes	ComputationalTime (s)	Accuracy
2	1	275	83.80%
5	1	337	84.06%
10	1	416	84.13%
15	1	532	84.10%
20	1	757	83.90%

Figure 15: Tensorflow : Readings for single node

numTrees	numNodes	ComputationalTime (s)	Accuracy
2	2	277	83.80%
10	2	397	83.90%
15	2	431	84.10%
20	2	522	84.10%
30	2	709	84.10%

Figure 16: Tensorflow : Readings for two nodes

In figure 17, we observe that the computational time increases almost linearly with two nodes as the number of trees is increased. However, for the one node, the increase almost exponential and

it diverges significantly from the time for two nodes. It can be concluded that with tensorflow a significant gain in performance occurs soon after 5 trees. That is the gain overtakes the overhead of communication between executors.

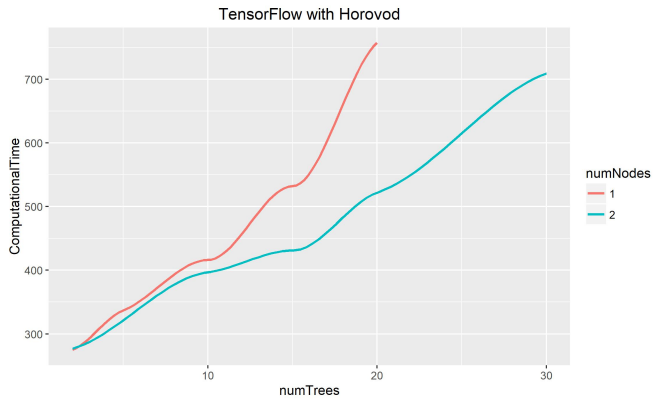

Figure 17: Tensorflow : Time versus Number of Trees

In figure 18, the accuracy for two node implementation plateaus after 15 trees.Also the variation is because of the randomness involved in building the trees. Also, it should be noted that the scale of the varying is at very minute scale and is zoomed in perspective.

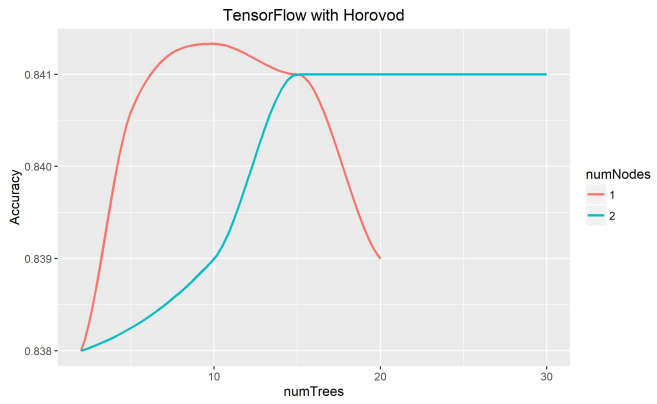

Figure 18: Tensorflow : Accuracy versus Number of Trees

5.5 Challenges

- One of the major challenges we faced was the installation of Horovod. Since the Juliet node did not have GPU and the default installation of Horovod searches for a GPU we needed to tweak certain files to get it working on the Juliet nodes.
- Another major drawback with using horovod is that it fails after a certain maximum process that can be spawned. We observed that for 30 and beyond the one node implementation failed and 40 and beyond failed for two node implementation.

6 PERFORMANCE COMPARISON OF SPARK AND TENSORFLOW WITH SINGLE AND MULTIPLE NODES

We observed that the time taken by tensorflow was on the higher side in comparison with the time taken by spark for both single and two nodes.

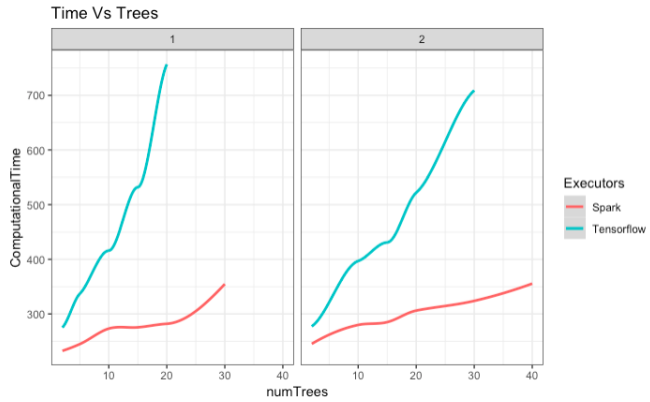

Figure 19: Time versus Number of Trees for Spark and Tensorflow

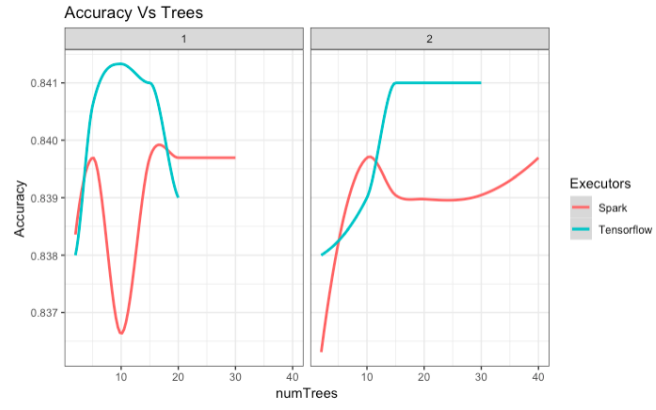

Figure 20: Accuracy versus Number of Trees for Spark and Tensorflow

From the above plot (fig 20) we can see that the accuracy of both varies in a scale of 0.836 to 0.842 together. We could not take more observations as the job failed for tensorflow for trees more than 20 and 30 for a single node and two node implementation respectively. But from the observations we have, we can see that there is a bit of higher accuracy achieved by tensorflow.

7 CONCLUSION

Consolidated results so far.
Note the tabulation is for 10 trees,

Approach	F1Score	Computation time (s)
Sequential approach	0.81157	886.382
Parallel approach	0.81198	198.502
Parallel with modified splitting criteria	0.83897	232.465

Spark: Parallelization of the forest decreased the computation time by a factor of 4. Modifying the splitting criteria allowed to improve the accuracy by 3.5%.

Approach	F1Score	Computation time (s)
Sequential approach	0.84157	1380
Parallel approach	0.8413	416
Parallel with two nodes	0.8392	397

Tensorflow: Parallelization of the forest for tensorflow decreased the computation time by a factor of 3.3.

In both spark and tensorflow the reduction in time was on a factor of 3 to 4.

8 ACTIVITIES

8.1 Revised Responsibilities

1. Random Forest Implementation: Sumeet and Srinithish
2. Spark Installation: Sumeet
3. Spark Implementation on single node : Srinithish
4. Scaling spark to multi nodes: Sumeet
5. Visualization of runs with Spark web UI for single node:Sumeet
6. Visualization of runs with hadoop web UI for two nodes:Sumeet
7. Installing Tensorflow and Horovod :Srinithish
8. Tensorflow Implementation of random forest: Sumeet
9. Scaling Tensorflow to multi nodes: Srinithish
10. Comparing performance between them:Sumeet and Srinithish

9 REFERENCES

(1) http://docs.h2o.ai/h2o/latest-stable/h2o-docs/data-science/drf.html
(2) http://cs229.stanford.edu/ proj2005/AziziChaiChui- DistributedRandomForests.pdf
(3) https://spark.apache.org/
(4) https://www.tensorflow.org/
(5) http://www.adaltas.com/en/2018/05/29/spark-tensorflow-2-3/
(6) Q. Tao, D. Chu, and J. Wang, âĂIJRecursive support vector machines for dimensionality reduction,âĂİ Neural Networks, IEEE Transactions on, vol. 19, no. 1, pp. 189 to 193, January 2008.
(7) Y. Lin, T. Liu, and C. Fuh, âĂIJMultiple kernel learning for dimensionality reduction,âĂİ Pattern Analysis and Machine Intelligence, IEEE Transactions on, vol. 33, no. 6, pp. 1147 to 1160, June 2011.
(8) C. Strobl, A. Boulesteix, T. Kneib, and T. Augustin, âĂIJConditional variable importance for random forests,âĂİ BMC Bioinformatics, vol. 9, no. 14, pp. 1 to 11, 2007.
(9) S. Bernard, S. Adam, and L. Heutte, âĂIJDynamic random forests,âĂİ Pattern Recognition Letters, vol. 33, no. 12, pp. 1580âĂŞ1586, September 2012.
(10) T. M. Khoshgoftaar, J. V. Hulse, and A. Napolitano, âĂIJComparing boosting and bagging techniques with noisy and imbalanced data,âĂİ Systems, Man and Cybernetics, IEEE Transactions on, vol. 41, no. 3, pp. 552âĂŞ568, May 2011.
(11) G. Yu, N. A. Goussies, J. Yuan, and Z. Liu, âĂIJFast action detection via discriminative random forest voting and top-k subvolume search,âĂİ Multimedia, IEEE Transactions on, vol. 13, no. 3, pp. 507 to 517, June 2011.
(12) G. Biau, âĂIJ Analysis of a random forests model,âĂİ Journal of Machine Learning Research, vol. 13, no. 1, pp. 1063 to 1095, January 2012.
(13) J. D. Basilico, M. A. Munson, T. G. Kolda, K. R. Dixon, and W. P. Kegelmeyer, âĂIJComet: A recipe for learning and using large ensembles on massive data,âĂİ in IEEE International Conference on Data Mining, October 2011, pp. 41 to 50.
(14) A Parallel Random Forest Algorithm for Big Data in a Spark Cloud Computing Environment Article in IEEE Transactions on Parallel and Distributed Systems Âů August 2016.
(15) Random Forests for Big Data RobinGenuera, Jean-MichelPoggib, ChristineTuleau-Malotc, NathalieVilla-Vialaneixd.