Random Forest for Big Data

Jainendra Kumar
School of Informatics and Computing
Indiana University
Bloomington, United States
jaikumar@iu.edu

Gattu Ramanadham
School of Informatics and Computing
Indiana University
Bloomington, United States
ramgattu@iu.edu

Surya Prateek Soni
School of Informatics and Computing
Indiana University
Bloomington, United States
susoni@iu.edu

ABSTRACT
Over the last ten years, popularity in MapReduce and large-scale data processing has led to the emergence of wide variety of cluster computing frameworks. With rich language-integrated APIs and wide range of libraries Apache Spark has become the de facto open source big data processing framework. Along with big data, comes machine learning libraries for differential programming on huge datasets. One of the most popular libraries is TensorFlow introduced by Google Brain team. Focusing on big data classification problem, our project aims to do a selective review and performance comparison through experiments by analyzing distributed implementation of Random Forest algorithm on Apache Spark and sequential Random Forest algorithm using machine learning libraries of TensorFlow.

General Terms
Algorithms, Measurement, Documentation, Performance, Design, Reliability, Experimentation.

Keywords
Random Forest, Apache Spark, TensorFlow, Big Data, Machine Learning

1. INTRODUCTION
Both Apache Spark and TensorFlow support distributed computing and is able to work across a cluster of computing nodes. However, we need to understand which framework to use for what computational task. Earlier, Spark was a standalone framework for cluster computing but with the release of Spark MLlib (Spark's machine learning library) it has made practical machine learning easy, fault tolerant and scalable. At a very high level, it provides tools for ML Algorithms, Featurization, Pipelines, Persistence and Utilities. TensorFlow on the other hand, improves the performance of numerical computation without the requirement of implementing complex learning algorithms via its own APIs and libraries.

So to understand the functioning of both the frameworks we would want to take a single classification problem based on decision trees and implement that on both the frameworks so as to understand the performance, accuracy and overhead of the two tools. We then propose a benchmarking framework analyzing the overall compute performance of the two.

Last year, yahoo released a new project called TensorFlowOn-Spark, a pairing of the two frameworks. In future, we would like to shed some light on why and when the pairing of the two tools would become more beneficial to researchers and engineers.

1.1 Related Work
In the work of Wright, et. al. have introduced a "Fast implementation of Random Forests for High Dimensional Data in C++ and R". They have introduced C++ application and R package ranger which is implemented in C++ and uses standard libraries. Extensive use of parallel processing is implemented with the help of thread library. In simulation studies it has been proved that computational and memory efficiency of Ranger outperformed any execution of Random Forest so far and is the fastest implementation of Random Forest for high dimensional data.

High Performance Big Data is one of the major challenges for statistical science. In the work of Geneur, et. al. "Random Forest for Big Data" have given a selective review of available proposals that take care of scaling Random Forest to Big Data problems. They have also formulated and compared various versions of Random Forest namely sequential RF, sampling RF, parallel RF, m-out-of-n RF, Bag of Little Bootstraps RF, divide and conquer RF and online RF in Big Data context.

1.2 Spark
Spark works on master/worker architecture. It has a layered architecture where all layers are loosely coupled and are connected via API extensions.

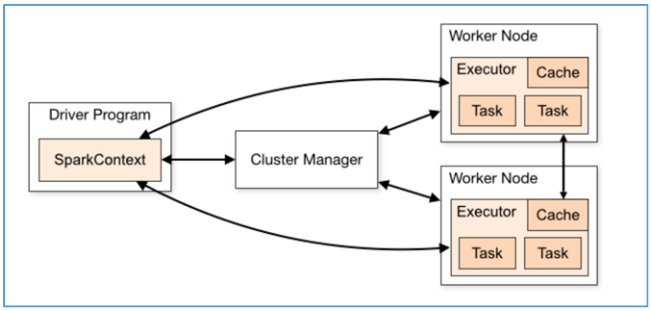

Fig 1. Spark Architecture (https://spark.apache.org/docs/latest/cluster-overview.html)

A spark cluster has single master and multiple slaves/workers. These can run on same spark cluster or on multiple slave machines.

Spark driver runs the main function of the spark program and is responsible for creating a spark context. It consists of DAG-Scheduler, Task Scheduler, Backend Scheduler and Block Manager which converts spark code into actual spark jobs.

When a user submits a spark code, the driver converts the code into a logical directed acyclic graph (DAG). The driver also performs pipelining transformations which is when the directed acyclic graph is converted to an execution plan with logical execution steps.

1.3 TensorFlow
TensorFlow consists of a C API which abstracts the system level code from the user level code implemented in different languages. It consists of four main layers: Client, Distributed Master, Worker Services and Kernel Implementations.

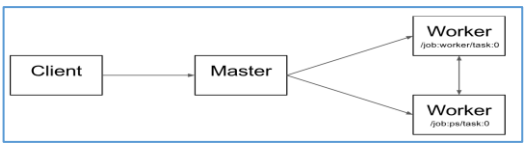

Fig 2. (https://www.tensorflow.org/images/diag1.svg)

Client maps the computation that needs to be performed as a dataflow graph and it initiates a graph execution using a session. Master is responsible for splitting the graph into a subgraph and then assigns the subgraph representing different tasks to worker services. Worker services implement the execution of graph operations using kernel implementations depending upon the hardware selected (CPU or GPU). They can also communicate with other worker services.

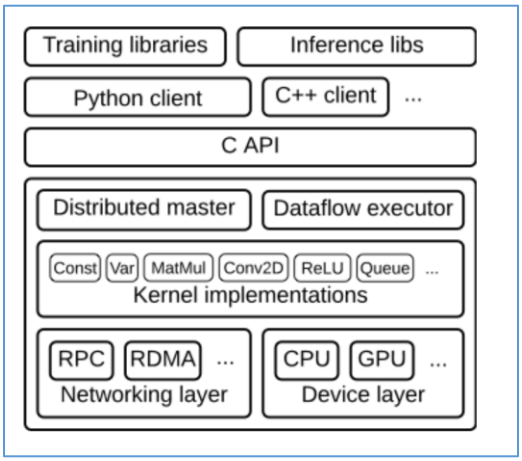

Fig 3. (https://www.tensorflow.org/images/layers.png)

Kernel executes the implementation of specific graph operations. This is usually implemented using Eigen::Tensor, which uses C++ template at machine level to generate code specific to chosen hardware whether CPU or GPU.

2. RANDOM FOREST

Random Forest introduced by Breiman [1] are a popular group of classification and regression methods which are also called as Ensemble method. They use divide-and-conquer approach to improve performance. A random forest starts by creating multiple decision trees. In a single decision tree an input is entered from the top and as the tree is traversed down the input it gets classified into smaller and smaller datasets thereby at the root giving a single predicted classification. Random Forest algorithms are widely used [3] [4] and their consistency has been demonstrated by Scornet et al. [5]

Since, Random Forests are composed of several decision trees it seems very obvious that they can be implemented in parallel and a fast implementation of Random Forest can easily be obtained. However, direct parallel training of random forest is not at all practical due to huge size of datasets. Hence they involve multiple bootstrapping schemes to handle massive datasets, in addition to performing parallel processing.

In this paper we experiment on the methods that have already been adopted for implementing Random Forest on distributed environment for large datasets. We experiment two (sequential and parallel) variants of Random Forest on a real-world large dataset of 1 million observations.

2.1 Algorithm

Random Forest are ensembles of decision trees. From the original dataset the algorithm generates k different data subsamples using bootstrapping methodology and then k decision trees are formed once the subsamples are trained. A random forest is thus a collection of these trained decision trees. Each decision tree is used to classify a test sample and the final classification is the result of the majority votes of individual decision trees.

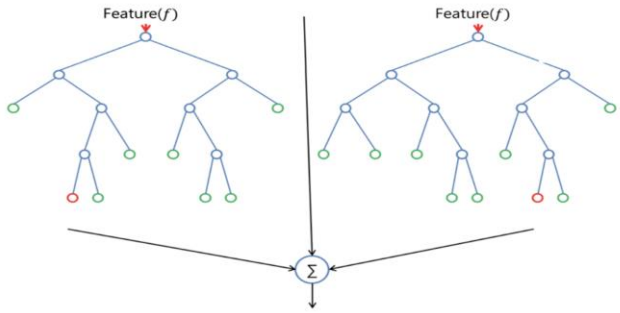

Fig 4 .(https://towardsdatascience.com/the-random-forest-algorithm-d457d499ffcd)

The pseudocode of the algorithm is as follows:

1. Randomly select “f” features from total of “n” features where f < n
2. From the selected “f” features, calculate the node “d” which has the best split point.
3. Split the node into child nodes using the best split.
4. Repeat steps 1 to 3 until “l” number of nodes have been reached.
5. Build forest by repeating steps 1 to 4 to form “n” number of trees.

The major challenge in decision tree creation is to decide which node should be considered as the root node. Thus, we have different selection measures to determine which can be considered as the root node. The two main popular attribute selection measures are information gain [5] and gini index [6].

We can calculate the information gain by measuring the entropy measure of each attribute which is the measure of the randomness of that particular attribute. It is given by:

$\sum Ci=1-filog(fi)$ where fi is the frequency of label i at a node and C is the number of unique labels. For example, for binary classification problem with value as 0 and 1, if all examples are 0 or all examples are 1 then the entropy will be zero or low. If equal number of records are zeroes and ones then the entropy will be one. Therefore, the attribute with higher entropy values are kept as root while attributes with lower entropy values are pushed to the root.

Gini index on the other hand is a metric which identifies how often a randomly chosen attribute would be wrongly classified. It is given by:

$\sum Ci=1fi(1-fi)$ where fi is the frequency of label i at a node and C is the number of unique labels. Therefore, it suggests that an attribute with lower Gini index should be preferred.

Overfitting is a major problem in when building a decision tree classifier. Overfitting is considered when the decision tree starts to become deeper and deeper so as to reduce the training set error but in turn it gets compromised with increase in test set error. This generally eliminated by performing pre-pruning and post-pruning techniques.

2.2 Dataset

For our random forest implementation, we take the 2006 airlines data provided by the Bureau of Transportation Statistics with 1 million unique observations. Some of the important attributes of the dataset that are relevant to our experimental results are denoted below.

Departure Performance

Attribute	Description
CRSDepTime	CRS Departure Time
DepTime	Actual Departure Time
DepDelay	Difference in scheduled and actual departure.
DepDel15	Departure Delay 15 Minutes or More (1=Yes)
DepartureDelayGroups	Departure Delay intervals
DepTimeBlk	CRS Departure Time Block, Hourly Intervals
TaxiOut	Taxi Out Time, in Minutes
WheelsOff	Wheels Off Time (local time: hhmm)

Arrival Performance

WheelsOn	Wheels On Time
TaxiIn	Taxi In Time, in Minutes
CRSArrTime	CRS Arrival Time
ArrTime	Actual Arrival Time
ArrDelay	Difference in scheduled and actual arrival
ArrDel15	Arrival Delay Indicator
ArrivalDelayGroups	Arrival Delay intervals
ArrTimeBlk	CRS Arrival Time Block, Hourly Intervals

Table 1.

3. PLATFORM

Our experiments are performed on FutureSystems cluster and Amazon Web Services (AWS) cloud with an aim to do performance comparisons of both the platforms.

3.1 FutureSystems cluster

"Juliet" cluster is a High-Performance Computing cluster provided by FutureSystems. It is a part of NSF's high-performance cyberinfrastructure and led by Indiana University. Its test bed is composed of a high-speed network, connecting distributed clusters of high-performance computers that employ virtualization technology to support different operating systems.

3.2 Amazon Web Services (AWS)

We have also used AWS compute infrastructure in our experiments which is denoted as Amazon elastic compute cloud (ec2). It provides secure, resizable, on-demand compute capacity in the form of OS instances which are geographically distributed across the world and connected by Amazon's backbone network.

4. APPROACH

Our first approach was to clean the data and make it suitable for our algorithm. The 2006 airlines data consists of 29 features with datatypes in Strings, Integer and Float. There are approximately 1 million unique records in the data. The data cleaning was performed by removing specific individual records that contained missing values, Null or NaN values and incoherent values.

Our main problem statement in Random Forest algorithm was to predict and classify the delay status of each flight. In other words, the algorithm will predict whether the flight will be delayed or not by giving binary classification of "0" and "1" to each record where zero denotes the flight is predicted as on time while one denotes the flight is predicted as delayed. This classification was then matched with the actual correct data result to predict the classification accuracy.

After the initial data cleaning, our first step was to construct the correct labeling for each data entry. Each data entry was classified into two classification values (yes and no) denoting the flight is delayed or not, yes denoting the flight is delayed and no denoting it is not. This is performed based on the entry in the "DepDelay" feature which denotes the departure delay. If the "DepDelay" is greater than zero then that flight data entry is classified as delayed otherwise it is classified as on time.

The algorithm then itself converts "yes" and "no" to binaries of 0s and 1s to perform computation. The data is then split into training data and test data in the ratio of 70 to 30. The algorithm trains the model on 70 percent of data while testing is done on 30 percent of data. In the training as well as the testing phase the "DepDelay" feature is not passed to the algorithm since that feature was used to label the data entries. Once, the cleaning and labeling is completed we inputted the data into the algorithm and ran it utilizing Apache Spark's Mllib library and TensorFlow's TensorForest estimator.

4.1 Mllib for Apache Spark

We make use of spark.ml implementation of random forest algorithm. spark.mllib library supports random forest for both binary and multiclass classification where we can use both continuous and categorical features. spark.mllib implements random forests using the existing decision tree implementation. It injects randomness into the training process by subsampling technique or bootstrapping such that on each iteration a different training data is obtained [9]. It also considers different random subsets of features to split on at each node [10].

On algorithmic level, some of the tunable parameters defined by the mllib library that we have experimented with are:

maxDepth: Maximum depth of a tree.
minInfoGain: The minimum information gain for a further split to take place.
maxBins: Number of bins used when discretizing features.
impurity: Impurity chosen e.g. gini.

On platform level some of the tunable parameters for running Spark jobs that we have experimented with are:
executor-cores: number of concurrent tasks an executor can run.
num-executors: number of executors requested.
executor-memory: heap size of the spark job.

4.2 TensorForest for TensorFlow

Tensor Forest estimator is an high level API, which greatly simplifies machine learning. They encapsulate training, evaluation and prediction.

Estimators are provided custom or premade, for this project we have used pre made estimators. Estimators models on local host or on a distributed server environment without changing model. Tensors are the central unit of data in tensorflow. Tensors are like arrays, we can conceptually think of them as n-dimensional matrices. These tensors are passed to operations that perform computations on them.

4.3 Experimental Results

4.3.1 Sequential Implementation on Tensor Flow and Spark

We first compare the sequential execution of Random Forest on Spark and Tensorflow on a single node Juliet Cluster.

maxDepth	Execution Time in seconds	
	Spark	Tensorflow
5	43	306
10	54	410
15	61	503
20	73	583
25	89	601

Table 2.

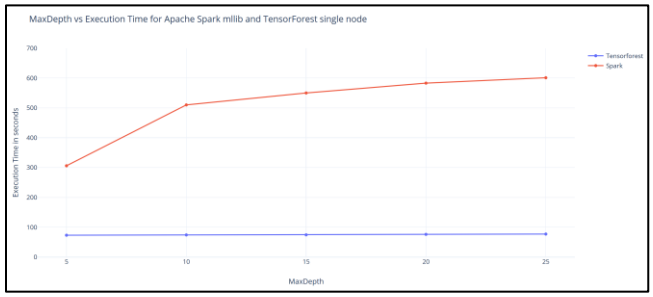

Figure 5.

In Figure 6. for measuring the performance comparison with respect to the number of Trees and time in seconds we keep the maxDepth constant as 10 and maxBins as 5.

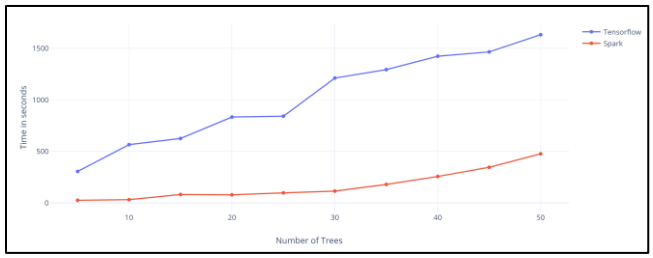

Figure 6. Number of Trees vs Time Taken in sequential execution

We can clearly see that the Spark's execution of Random Forest is faster than the Tensorflow's execution through TensorForest estimator when we give exactly the same parameters to both the classifiers on the same machine platform.

4.3.2 Parallel Implementation on Spark

For now, we have bench marked our algorithm by running it parallelly on a distributed two node cluster. We have implemented this via Spark by setting up a Hadoop Cluster on both Juliet Cluster and Amazon Web Services (AWS) cloud. We have then benchmarked and compared the performance of both the cluster and the cloud.

4.3.2.1 Algorithm specific results on Juliet Cluster

For measuring the accuracy of the algorithm with respect to the max depth of the Random Forest we have kept all other parameters as constant. The algorithm was tested while keeping the spark executors as 3, cores per executors as 15 and executor memory as 8 GB which is the ideal configuration with respect to our machine's capacity and the input data.

numTrees	Accuracy	Time in seconds	maxDepth	maxBins
50	73.51	89	1	5
50	73.65	220	5	5
50	74.46	302	10	5
50	74.58	451	15	5
50	76.24	601	20	5

Table 3.

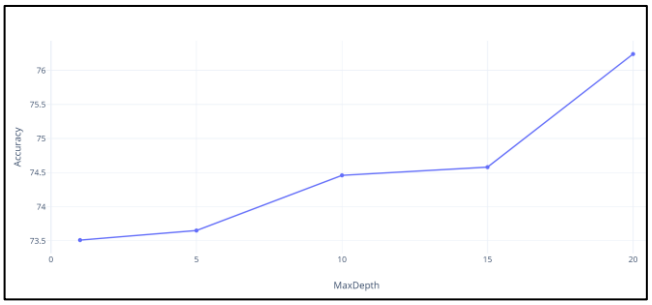

Figure 7. MaxDepth vs Accuracy

Therefore, by looking at the graph we can see a clear correlation between the accuracy and the max depth given to the algorithm. We assume that deeper trees characterized by higher max depth are more expressive and allow higher accuracy. But they are expensive to train, consume more execution and overhead time and likely to overfit.

However, an interesting correlation is observed by keeping the maxDepth as constant and then observing the relation between the number of trees and the accuracy. There is no significant changes in the accuracy as we saw in the previous graph where we changed the maxDepth.

numTrees	Accuracy	Time in seconds	maxDepth	maxBins
10	76.63	211	15	5
20	76.64	215	15	5
30	76.66	227	15	5
40	76.91	238	15	5
50	76.92	321	15	5

Table 4.

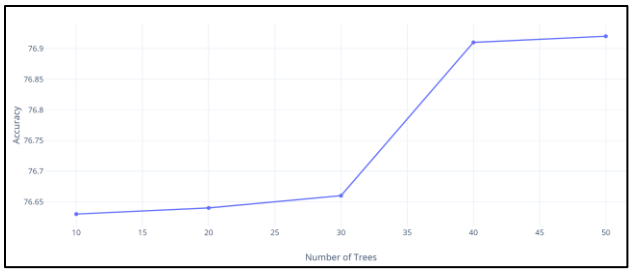

Figure 6. Number of Trees vs Accuracy

4.3.2.2 CPU specific results on Juliet Cluster

In this experiment we measure how our algorithm's execution time changes with respect to changing the number of spark executors and cores per executor. For this very purpose we keep the number of trees, depth and bins that will be used in the Random Forest algorithm as constant. We tune our algorithm with minimal set of parameters because we do not care about accuracy in this experiment. Hence, keeping the number of trees as 50, maxdepth as 5 and maxBins as 5 we get the following results.

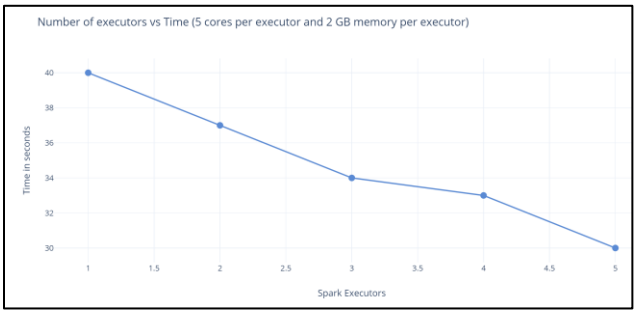

Figure 7. Number of Executors vs Execution Time

As we can see from the figure 6 as the number of spark executors are increased, keeping the number of cores per executor constant the execution time of our algorithm decreases. This is mainly because of the increase in parallelism in the spark job. However, we must also take into account that if the number of executors are increased, it takes more time for Spark to start the executors and run them, thus sometimes it is observed that the cost of overhead overshadows the reduction in actual application run time. To take this into consideration we have found the right balance between the cores per executor, memory per executor and the size of our data.

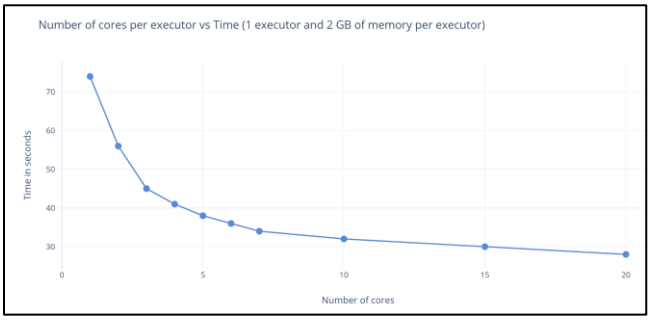

Figure 8. Number of cores vs Execution Time

Figure 7 shows the performance of the algorithm in terms of the execution time with respect to the number of cores per executor. We have again tried to find the right balance between the size of the data and number of cores by keeping the executors and memory per executor as constant. For our specific application we found that keeping 1 executor and 2g memory per executor would be an ideal scenario to measure the relation between the number of cores and the execution time. As observed, as the number of cores are increased the execution time of the algorithm decreases. We conclude that increasing the number of cores allows for more number of concurrent tasks which can be executed on the machine with a single executor. It has also been observed that increasing the number of cores also reduces the memory overhead of the executor.

4.3.3. Performance comparison of Juliet Cluster and Amazon Web Services (AWS) cloud

We also experimented by implementing the distributed Random Forest algorithm on Amazon's ec2 compute cloud. This was done by setting up the 2 node Hadoop cluster including setting up the name nodes and secondary name nodes on AWS on top of Hadoop installed on Ubuntu Linux ec2 instance.

We performed benchmarking by increasing the depth from 5 to 20 on minimal configuration of 1 GB memory per executor and 1 core per executor, and 2 executors. With our initial results we found that the Spark's performance on t2.medium (2vCPU, 4G RAM) was not as good as on the cluster owing to high latency in ec2 instances located in multiple availability zones and less compute power for spark jobs.

Trees	Depth	Time (seconds) AWS	Time (seconds) Juliet Cluster
10	5	88	55
10	10	119	59
10	15	138	72
10	20	176	94

Table 5.

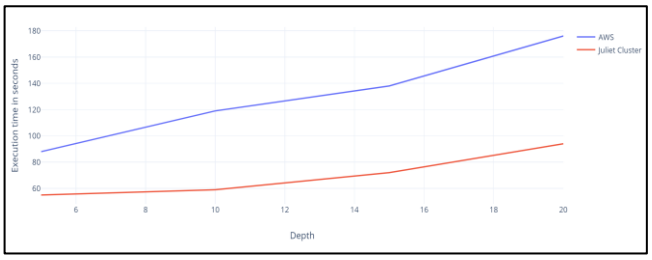

Figure 9. AWS vs Juliet Cluster

We conclude that, this experiment is not a very good indication of Spark's performance on cloud because of constraints in available memory and CPU requirements on AWS as comparison to Juliet Cluster which may require further investigation and experiments. However, with some initial results we suggest that the Juliet Cluster performs better than cloud owing to its high number of virtual cores, high RAM and very low data latency during cross node communication.

5. CONCLUSION

We implemented a popular classification algorithm – Random Forest on real world data and showed how it can be scaled in a distributed environment using Apache Spark. We also showed and benchmarked the sequential performance of the algorithm by comparing the performance of Apache Spark's mllib library and Tensorflow's tensorforest in terms of execution time and accuracy.

With our initial results we conclude that MLlib implementation of Random Forest is faster than the Tensorforest estimators owing to the in-memory computation of Spark's executors. Transformations are very powerful characteristic of Spark. It comes with very

advanced Directed Acyclic Graph (DAG) or a data processing engine that is more suited for classification problems. However, unlike TensorFlow, Spark does not have any out of the box support for deep neural network architectures such as Convolutional Neural Network for images and Recurrent Neural Network.

We also conclude that although we are yet to test the distributed implementation of Random Forest on Tensorflow but we assume that Tensorflow would be more efficient in terms of execution time if it runs on GPU architecture. Also, having a GPU of clusters would be an expensive strategy for running tasks which can be implemented easily by Apache Spark. For future work we would like to test TensorFlow's implementation of Random Forest in distributed way by executing it on multiple nodes.

6. ACKNOWLEDGEMENTS

The authors thank the professor Qiu and associate instructor Mr. Selahattin for their constant encouragement and help throughout the development of this project. The authors would also like to thank other students of ENGR E-599 for their suggestions, peer review and constructive criticism throughout the timeline of this project

7. REFERENCES

[1]L.Breiman,Randomforests,Mach.Learn.45(1)(2001)5-32,http://www.springerlink.com/content/u0p06167n6173512/fulltext.pdf.

[2]E.Scornet,G.Biau,J.Vert,Consistencyofrandomforests,Ann.Stat.43 (4)(2015)1716–1741,http://dx.doi.org/10.1214/15-AOS1321.

[3]A.Verikas,A.Gelzinis,M.Bacauskiene,Miningdatawithrandomforests:asur-veyandresultsofnewtests,PatternRecognit.44 (2)(2011)330–349,http://dx.doi.org/10.1016/j.patcog.2010.08.011.

[4]A.Ziegler,I.König,Miningdatawithrandomforests:currentoptionsforreal-worldapplications,WileyInterdiscip.Rev.DataMin.Knowl.Discov.4 (1)(2014)55–63,http://dx.doi.org/10.1002/widm.1114.

[5]Hastie, T., Tibshirani, R., and Friedman, J.The Elements of Statistical Learning. Springer, 10 edition, 2013.

[6]E.Scornet,G.Biau,J.Vert,Consistencyofrandomforests,Ann.Stat.43 (4)(2015)1716–1741,http://dx.doi.org/10.1214/15-AOS1321

[7]https://dataaspirant.com/2017/01/30/how-decision-tree algorithm-works/

[8] https://towardsdatascience.com/the-random-forest-algorithm-d457d499ffcd

[9][10]https://spark.apache.org/docs/2.2.0/mllib-ensembles.html#random-forests