Comparison of Random Forest Techniques over Distributed Frameworks

Rishab Nagaraj
School of Informatics and Computing
Indiana University, Bloomington
risnaga@iu.edu

Vivek Vikram Magadi
School of Informatics and Computing
Indiana University, Bloomington
vmagadi@iu.edu

ABSTRACT
This project deals with making a comparison of the performance displayed by running single and multiple node implementations of the Random Forest Algorithm using Spark and TensorFlow.

General Terms
Algorithms, Performance, Experimentation, Theory.

Keywords
Spark, TensorFlow, Random Forest, Physical Activity.

1. INTRODUCTION
This project proposes a comparative study of different techniques (Spark, TensorFlow) that could be used to effectively parallelize the Random Forest algorithm.

The Random Forest algorithm is an ensemble learning method for classification, regression and other tasks that operates by building many decision trees during training and either returning the class that occurs the most out of all the classes (classification) or returning some mean prediction (regression) of the individual trees.

The algorithm was trained on the Physical Activity Monitoring (PAMAP2) dataset where it predicted a person's physical activity based on different features.

Physical activity or exercise is said to improve a person's health and minimize the risk of contracting diseases like diabetes, cancer, cardiovascular disease, etc. It can have immediate and long-term health benefits and most importantly, regular activity can improve quality of life.

Recent technological advancements have made it easier to track a person's physical activity through the use of smartwatches and other wearable devices. These devices are widely used and have, therefore, generated a lot of physical activity data for different people around the world. Analyzing a person's physical activity could be useful in determining their fitness levels and possibly lead to early recognition of symptoms of diseases.

2. DATASET
The dataset used is the Physical Activity Monitoring (PAMAP2) [1] dataset that contains data of 18 different physical activities (walking, cycling, driving, sitting, etc.), performed by 9 subjects, each wearing 3 inertial measurement units (IMUs) i.e., one on each hand, one on the chest and another on the ankle as well as a heart rate monitor. The IMUs record temperature and acceleration among other things. Each subject had to follow a protocol of 12 activities. In addition to this, some of the subjects also performed a few optional activities. One line in the dataset corresponds to one timestamped and labelled instance of sensory data. Each row has 54 columns that correspond to the timestamp, activity label and 52 attributes of the sensor data.

The sensor data is split as heart rate, hand sensor data, chest sensor data and ankle sensor data. Each sensor records 17 attributes.

3. ARCHITECTURE
Spark [2] is an open-source cluster-computing framework that is built around speed, ease of use and streaming analytics. Python includes a wide range of libraries and is used for Machine Learning and Real-Time Streaming Analytics. PySpark is a Python API for Spark that allows a combination of Python's simplicity and Spark's power to utilize Big Data effectively.

Resilient Distributed Datasets (RDDs) form the foundation of a Spark application. They are fault tolerant and capable of rebuilding data on failure. The data is distributed among the multiple nodes in the cluster. RDDs allow for transformation and action operations which create new RDDs and perform computations respectively.

Spark was developed in 2012 to counter limitations in the MapReduce cluster which forced linear dataflow. Spark's RDDs function as a set of distributed programs that offer a restricted distributed shared memory. Spark facilitates implementation of iterative algorithms and exploratory data analysis. The latency of applications can be reduced by several orders of magnitude when compared to MapReduce.

Spark requires a cluster manager and a distributed storage system. For these, Spark supports Hadoop YARN or Apache Mesos and HDFS, Amazon S3, Cassandra or Kudu.

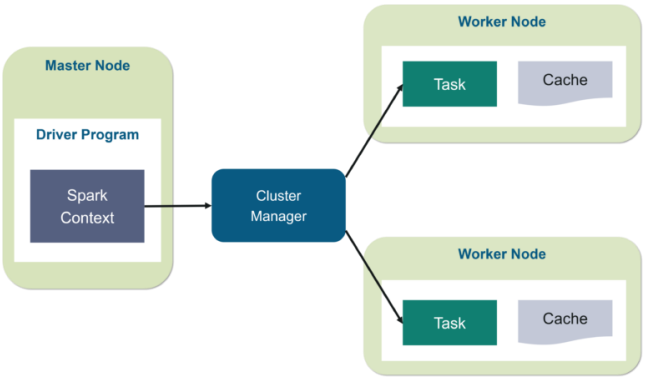

Figure 1: Spark Architecture

The Master Node [3] holds the Driver Program within which, you create Spark Context that is the gateway to all Spark functionalities (like a database connection). Any command executed on Spark goes through the Spark Context. The Spark Context works with the Cluster Manager to manage various jobs. The contents of the Master Node handle job execution within the cluster. A job is split into multiple tasks, each of which are distributed over the Worker Nodes. If an RDD is created in the Spark Context, it can be distributed across various nodes and be cached there as well.

Worker nodes are slave nodes that execute given tasks that are executed on partitioned RDDs and the result is returned to the Spark Context. So, the Spark Context takes the jobs, breaks them into tasks and distributes them over the worker nodes. These tasks work on the partitioned RDD, perform operations, collect results and return them to the Spark Context. Increasing the number of workers allows for dividing the jobs into more partitions and can be executed in parallel over multiple systems which will result in the tasks executing faster. The increase in number of workers, can also increase memory size which will allow for caching jobs which will result in faster execution.

TensorFlow is a free and open-source software library for dataflow and differentiable programming across tasks. It is used for machine learning applications and used for research and production at Google. Its flexible architecture allows for easy deployment of computation across various platforms as well as from desktops to server clusters to mobile devices.

TensorFlow computations are expressed as stateful dataflow graphs. TensorFlow derives its name from the operations that neural networks perform on multidimensional data arrays (tensors).

TensorFlow [4] is designed for large scale distributed training and inference but is also flexible enough to support experimentation with new machine learning models and system level optimizations.

The client program can directly compose individual operations to build neural network layers and other high-level abstractions. The client creates a session which sends the graph definition to the distributed master as a protocol buffer. When nodes in the graph are evaluated, a call is made to the distributed master to initiate computation.

The distributed master prunes the graph to obtain the subgraph necessary for evaluating the nodes requested by the client. It partitions the graph to obtain subgraphs for each device as well as caches these subgraphs for reuse in later steps. The master sees the overall computation for a step and applies standard optimizations (common subexpression elimination and constant folding). It then coordinates execution of the optimized subgraphs across a set of tasks.

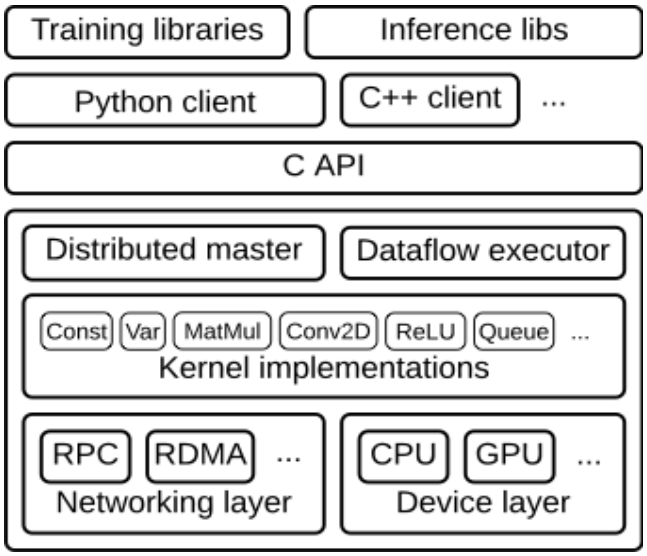

Figure 2: TensorFlow Architecture

The worker service present in each task handles requests from the master, schedules execution of kernels for operations that comprise a local subgraph and mediates direct communication between tasks. The worker service is optimized to run large graphs with low overhead. It dispatches kernels to local devices and runs parallel kernels when possible.

4. RELATED WORK

Random Forests by Leo Breiman [5] introduced the concept of Random Forests as a combination of tree predictors where each tree depends on values of a random vector sampled independently and with the same distribution for all trees in the forest. Generalization error depends on the strength of individual trees in the forest and the correlation between them. Error rates obtained by using a random selection of features to split nodes are comparable to those obtained by Adaboost but are more robust. Estimates of monitor error, strength and correlation are used to show a response to increasing the number of features used to split. These can also be used to measure variable importance.

Random Forests for Big Data by Genuer and Poggi et al. [6] focuses on the classification problems and proposes a selective review of available proposals that deal with scaling random forests to Big Data problems. These proposals rely on parallel environments or on online adaptations of random forests. Experimentation is done on five variants of two massive datasets (around 15 and 120 million observations). One variant relies on subsampling while three others are related to parallel implementations of random forests and involve either various adaptations of bootstrap to Big Data or to "divide-and-conquer" approaches. The fifth variant relates on online learning of random forests. These numerical experiments lead to highlight the relative performance of the different variants, as well as some of their limitations.

A Parallel Random Forest Algorithm for Big Data in a Spark Cloud Computing Environment by Chen and Li et al. [7] presents a Parallel Random Forest (PRF) algorithm for big data on the Apache Spark platform. The PRF algorithm is optimized based on a hybrid approach combining data-parallel and task-parallel optimization. From the perspective of data-parallel optimization, a vertical data-partitioning method is performed to reduce the data communication cost effectively, and a data-multiplexing method is performed to allow the training dataset to be reused and diminish the volume of data. From the perspective of task-parallel optimization, a dual parallel approach is carried out in the training process of RF, and a task Directed Acyclic Graph (DAG) is created according to the parallel training process of PRF and the dependence of the Resilient Distributed Datasets (RDD) objects. Then, different task schedulers are invoked for the tasks in the DAG. Moreover, to improve the algorithm's accuracy for large, high-dimensional, and noisy data, we perform a dimension-reduction approach in the training process and a weighted voting approach in the prediction process prior to parallelization. Extensive experimental results indicate the superiority and notable advantages of the PRF algorithm over the relevant algorithms implemented by Spark MLlib and other studies in terms of the classification accuracy, performance, and scalability. With the expansion of the scale of the random forest model and the Spark cluster, the advantage of the PRF algorithm is more obvious.

TensorFlow: A System for Large-Scale Machine Learning by Abadi and Barham et al. [8] describes the TensorFlow dataflow model and demonstrates the performance that it achieves in real-world applications. TensorFlow uses dataflow graphs to represent computation, shared state and operations that can impact the state. Nodes of the dataflow graph are mapped across many machines in a cluster as well as within a machine across multiple computational devices (multicore CPUs, GPUs and TPUs). This architecture provides flexibility to the application developer. TensorFlow

enables developers to experiment with novel optimizations and training algorithms. It supports a variety of applications with a focus on training and inference on deep neural networks.

5. EXPERIMENTS

5.1 Preprocessing

After combining all user activity, we end up with 2,872,533 rows in our dataset. There are quite a few Null values, so we handle that by dropping columns with many Null values as well as rows that had Null values otherwise. After a training-test split of 70-30, this resulted in the dataset having a total of 1,844,568 rows.

5.2 Spark

5.2.1 Methodology

1. Combine all 9 different data sources and then apply preprocessing step.
2. Set up Spark Environment on Remote System.
3. Upload file to Hadoop File System
4. Run Spark job with default setting and record execution time.
5. Tune Spark job with different tuning parameters and record execution time.
6. Record multiple observations and take average of those observations.

5.2.2 Results

Spark required 3 parameters for execution, namely Number of Executors, Number of Cores per Worker and Memory used per Executor. Of these, the 2 parameters considered for tuning were:

1. Number of Executors – Varied from 2 to 44 in increments of 2.
2. Memory used per Executor – Varied from 2GB to 8GB in increments of 2GB.

The code was run on 88 combinations of the above tuning parameters for both single and multiple nodes and the performance was compared.

5.2.2.1 Keeping memory constant and changing number of executors

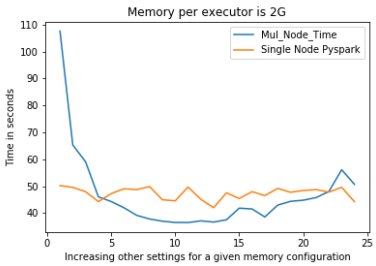

Figure 3: Memory = 2GB per Core

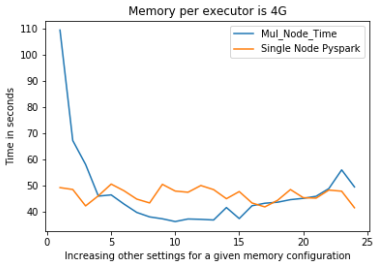

Figure 4: Memory = 4GB per Core

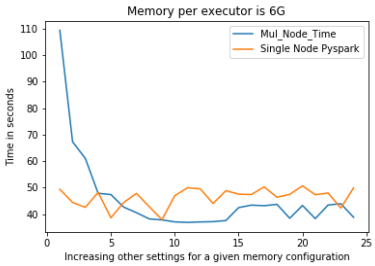

Figure 5: Memory = 6GB per Core

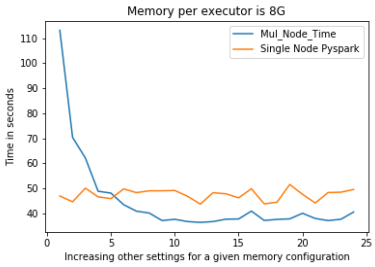

Figure 6: Memory = 8GB per Core

For most executors, we see that the time taken for the multiple node implementation is less than that of the single node implementation.

5.2.2.2 Keeping the number of executors constant and changing memory

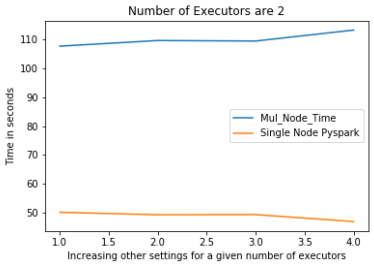

Figure 7: 2 Executors

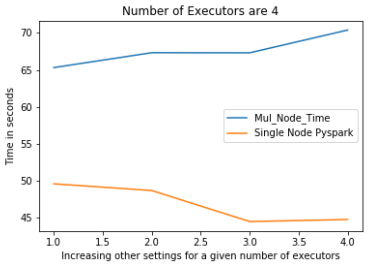

Figure 8: 4 Executors

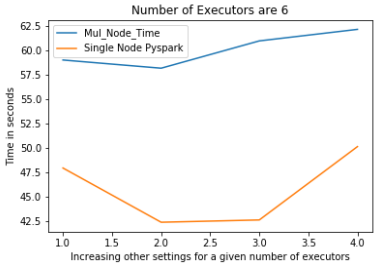

Figure 9: 6 Executors

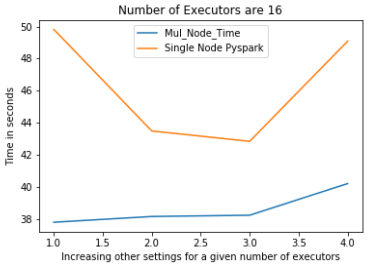

Figure 14: 16 Executors

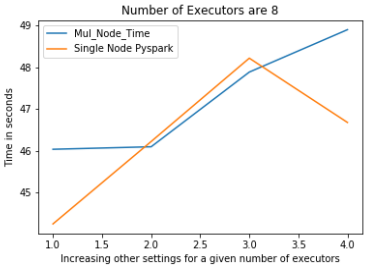

Figure 10: 8 Executors

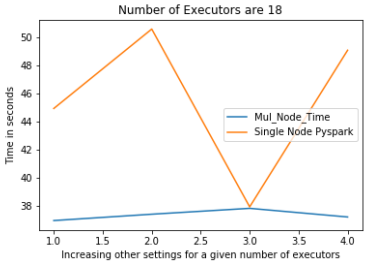

Figure 15: 18 Executors

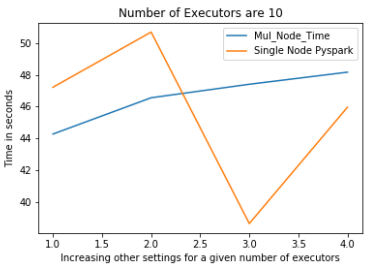

Figure 11: 10 Executors

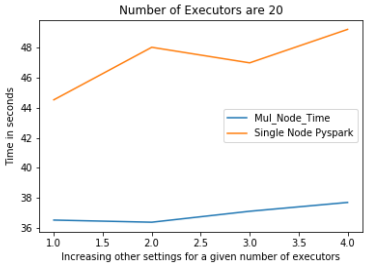

Figure 16: 20 Executors

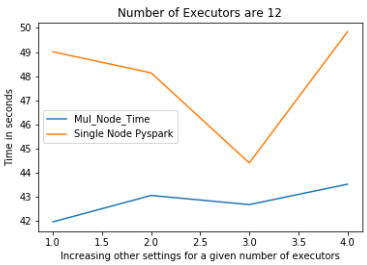

Figure 12: 12 Executors

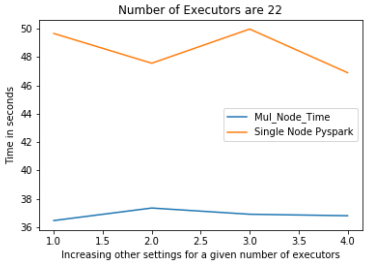

Figure 17: 22 Executors

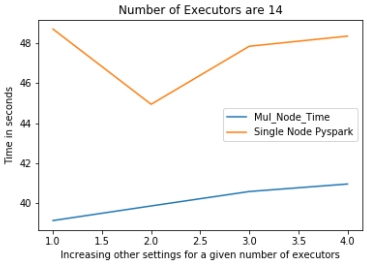

Figure 13: 14 Executors

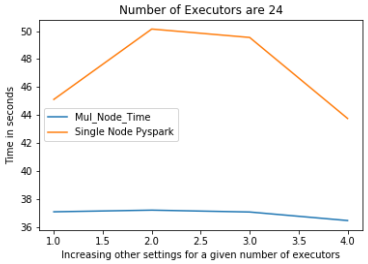

Figure 18: 24 Executors

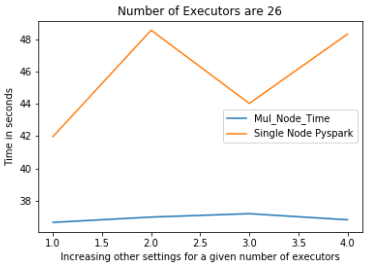

Figure 19: 26 Executors

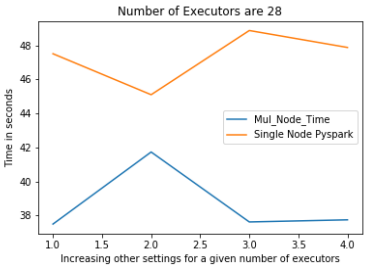

Figure 20: 28 Executors

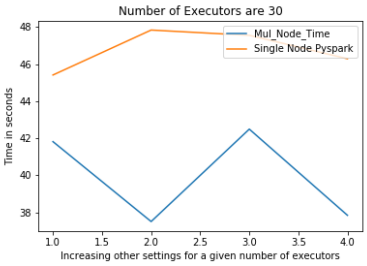

Figure 21: 30 Executors

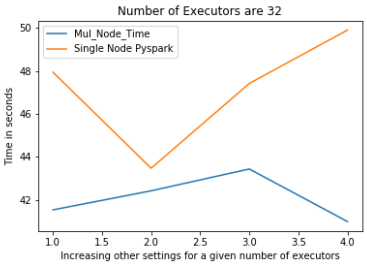

Figure 22: 32 Executors

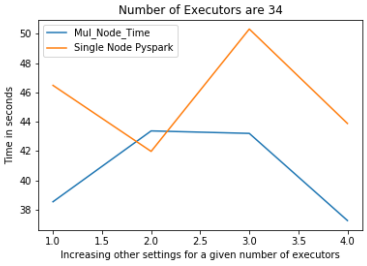

Figure 23: 34 Executors

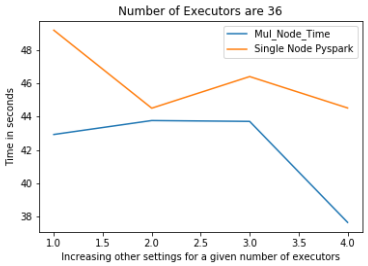

Figure 24: 36 Executors

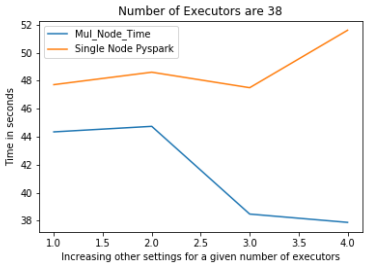

Figure 25: 38 Executors

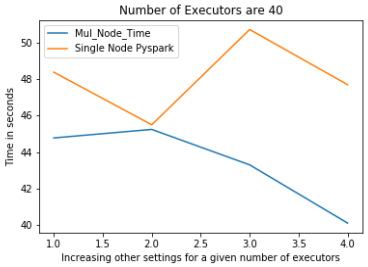

Figure 26: 40 Executors

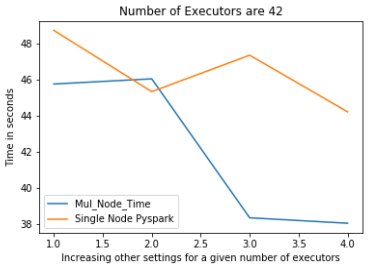

Figure 27: 42 Executors

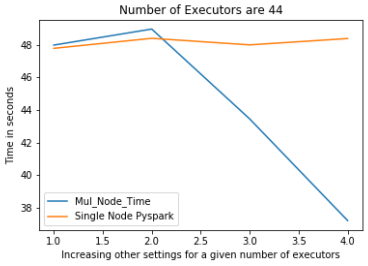

Figure 28: 44 Executors

We can see that from 12 Executors onward, only in certain rare cases, does the multi node implementation slow down when compared to single node.

5.2.2.3 *Increasing number of trees*

In the above experiments, the number of trees was kept as a constant i.e., 100 trees. We look at one configuration of executors and memory and vary the number of trees and observe the change in performance.

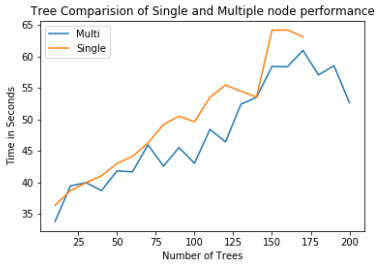

Figure 29: 38 Executors, 8GB per Core

As the number of trees used increases, we see an increase in time as well. Multiple node almost always performs better, as expected. In addition to performance, all these varying experiments gave an approximate accuracy of ~60%.

5.3 TensorFlow

5.3.1 *Methodology*

1. Combine all 9 different data sources and then apply preprocessing step.
2. Run the code with default setting and record execution time.
3. Tune the code with different tuning parameters and record execution time.
4. Record multiple observations and take average of those observations.

5.3.2 *Results*

The tuning parameter was the number of cores. The number of cores tested was 1, 2, 4, 8, 12, 24, 48.

5.3.2.1 *Changing number of cores*

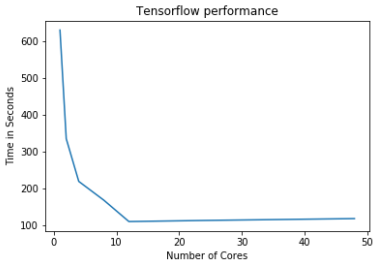

Figure 30: TensorFlow Performance over Number of Cores

From the graph, it is clear that, with the increase in cores, we see a sharp decrease in runtime that stabilizes at around 12 cores.

5.3.2.2 *Changing number of trees*

In the above experiment, the number of trees was kept constant at 10. We now vary the number of trees from 10 to 200 to observe the impact on performance.

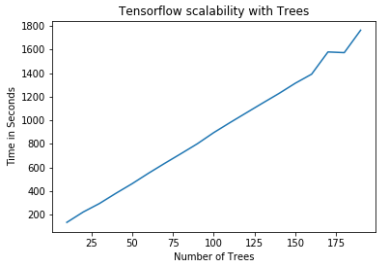

Figure 31: TensorFlow Performance over Number of Trees

It is clear that runtime linearly increases with the increase in number of trees. The accuracy obtained by running TensorFlow was ~70%

6. CONCLUSIONS

From all the experiments performed above, we see that Spark's runtime very rarely exceeds 1 minute (60 seconds). However, we see that TensorFlow very rarely executes in even close to 1 minute. Spark runs a lot faster than TensorFlow does. However, on looking at the accuracy, we begin to see a payoff i.e., Spark runs faster but TensorFlow is more accurate. As such, for the purposes of this project, we see that Spark is the better performer of the two.

7. ACKNOWLEDGMENTS

The authors would like to acknowledge and thank Professor Judy Qiu for the opportunity to work on a project of this nature as well as our mentor Selahattin Akkas for expertly guiding us every step of the way.

8. CONTRIBUTIONS

Rishab Nagaraj – Spark, Literature Survey

Vivek Vikram Magadi – TensorFlow, Visualizations

9. REFERENCES

[1] Reiss, A. and Stricker, D. 2012. Introducing a New Benchmarked Dataset for Activity Monitoring. *The 16th IEEE International Symposium on Wearable Computers (ISWC).*

[2] Chand, S. 2018. PySpark Programming – Integrating Speed With Simplicity. *Edureka Blog Post.* https://www.edureka.co/blog/pyspark-programming

[3] Vaidya, N. 2019. Apache Spark Architecture – Spark Cluster Architecture Explained. *Edureka Blog Post.* https://www.edureka.co/blog/spark-architecture/

[4] TensorFlow Architecture. *TensorFlow Guide.* https://www.tensorflow.org/guide/extend/architecture

[5] Breiman, L. 2001. Random Forests. *Machine Learning.* 45, 1, 5-32.

[6] Genuer, R. and Poggi, J. et al. 2017. Random Forests for Big Data. *Big data research.* 9, 28-46.

[7] Chen, J. and Li, K et al. 2017. A Parallel Random Forest Algorithm for Big Data in a Spark Cloud Computing Environment. *IEEE transactions on parallel and distributed systems.* 28, 4, 919-933.

[8] Abadi, M. and Barham, P. et al. 2016. TensorFlow: A System for Large-Scale Machine Learning. *The 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI).*