# Performance Evaluation of Distributed Machine Learning Platforms

Aditya Mukundan
*Dept of Electrical and Computer Engineering*
*Northeastern University*
mukundan.a@northeastern.edu

Bhanu Sai Simha Vanam
*Dept of Electrical and Computer Engineering*
*Northeastern University*
vanam.b@northeastern.edu

## *Abstract:*

**Machine learning has become increasingly popular across a wide range of application areas because of the ubiquity of big data and big computing. In recent times, dynamic research into several distributed computing platforms has plunged the world towards revolutionizing data processing in distributed platforms. This project aims to investigate the distributed machine learning platforms' architectural design because it inexorably influences the platforms' speed, scalability, and availability. Introducing TensorFlow in this project as an illustration of a more sophisticated dataflow system, and Spark as a representative dataflow system provides plethora of investigative data to study the given behavior of the system. We examine the communication and control bottlenecks for these methods from a distributed systems viewpoint. We also perform an investigative analysis on fault tolerance and development simplicity of the given system. This project aims to compare the performance of these two systems namely Spark and TensorFlow using two fundamental machine learning tasks—logistic regression and an example of image classification using the MNIST dataset—to provide a quantitative assessment.**

## I. MOTIVATION

With the revolutionary research and introduction to Artificial Intelligence, big data analytics and machine learning has made it necessary to have access to powerful computing resources that can handle large datasets. To meet this ever-increasing demand, distributed computing and machine learning platforms have emerged as crucial tools for data scientists and data architects. Distributed computing [1] refers to the use of multiple computers of partitions of memory that work in conjunction with each other to perform one or more tasks. This is often necessary when the task is too large or complex to be handled by a single computer. In distributed computing, the workload is divided into smaller tasks that can be performed simultaneously on multiple computers. This can greatly reduce the time it takes to complete the task and can improve the overall performance of the system. With performance, efficiency, scalability and time constraints as critical resources, distributed computing and distributed machine learning platforms have become crucial tools for data scientists and machine learning practitioners who need to handle large datasets and complex algorithms.

# II. INTRODUCTION

## *A. DATAFLOW SYSTEMS*

### *1. Spark:*

Spark is a widely used dataflow system for distributed computing that was initially developed at the University of California, Berkeley, and later became an open-source Apache project. Spark has gained popularity in recent years for its high-level programming interface and its ability to handle large-scale data processing and machine learning tasks.
Spark's dataflow engine allows users to express data processing tasks as a sequence of transformations on large datasets. These transformations can include operations like filtering, mapping, aggregating, and joining datasets. Spark's high-level programming interface, which includes APIs in Scala, Python, R, and Java, allows users to express these transformations in a concise and readable manner. Spark also offers a built-in SQL interface that allows users to perform SQL queries on datasets.

One of the key advantages of Spark is its ability to handle various data formats and sources. Spark can process structured data in formats like CSV, JSON, and Parquet, as well as unstructured data like text and streaming data. Spark's data processing engine can also handle graph data and machine learning tasks.

Spark is designed to be scalable and can run on clusters of computers. Spark's distributed architecture enables it to process large datasets by breaking them down into smaller partitions and distributing these partitions across a cluster of machines. Spark's data processing engine can then process these partitions in parallel, which enables Spark to handle big data processing tasks that would be challenging for a single machine.

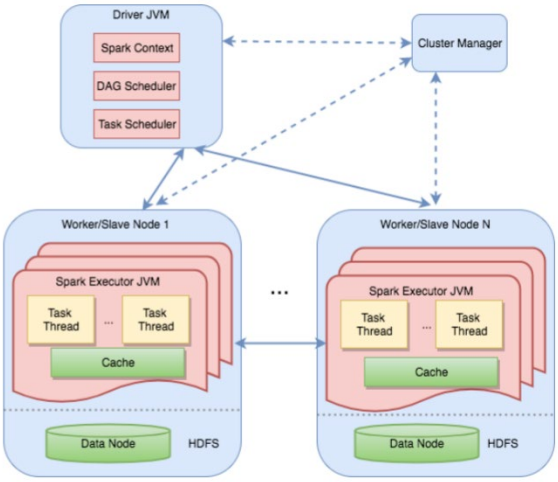


*Figure 1 : Spark Architecture [3]*

Spark is also designed to be fault tolerant. If a node in the cluster fails, Spark's engine can recover the lost data and continue processing the dataset without interruption. This feature is essential for long-running data processing jobs, where the likelihood of a node failure increases over time.

Another advantage of Spark is its support for in-memory data processing. Spark's engine can cache frequently accessed data in memory, which can significantly improve performance for iterative and interactive data processing tasks. Spark can also optimize data processing workflows for improved performance by reordering transformations and minimizing data shuffles between nodes.

Although Mllib[10], a machine learning library that includes standard machine learning algorithms, utilities, and linear algebra operations, has been retrofitted into Spark, which was not originally intended for machine learning. Spark stores the model parameters in the driver in the fundamental machine learning setup, and the workers interact with the driver to update the parameters after each iteration. The model parameters would need to be kept as an RDD for large-scale machine learning deployments, though, as they might not fit into the driver node.
Because a new RDD must be generated for the updated model parameters with each iteration, this adds a lot of overhead.

*2. TensorFlow:*

TensorFlow is a popular open-source machine learning framework developed by Google. It provides a flexible and efficient system for building and training machine learning models. One of the key features of TensorFlow is its dataflow system, which allows for efficient computation of complex operations on large datasets.

In a dataflow system, computations are represented as directed graphs, where nodes represent computations and edges represent data dependencies. In TensorFlow, these graphs are called computational graphs. The computational graph is created by defining the operations and the data inputs and outputs of the model.Once the computational graph is created, TensorFlow can optimize the computation by scheduling operations on devices (e.g., CPU, GPU) and by optimizing the memory usage. TensorFlow also supports distributed computing, where the computation is split across multiple devices, allowing for faster processing of large datasets.[9]

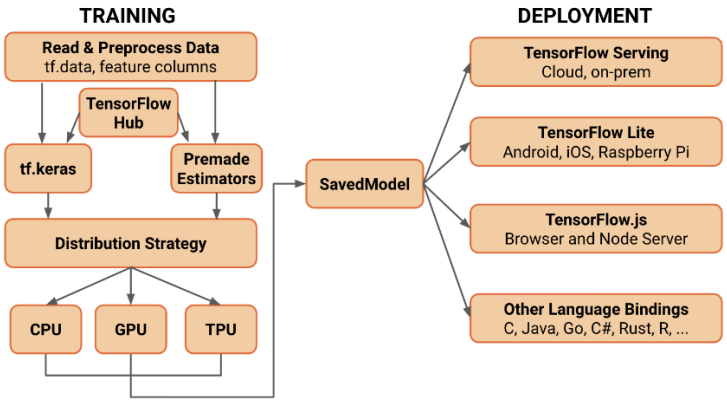


*Figure 2: TensorFlow Architecture*

TensorFlow uses a lazy evaluation strategy, which means that the computation is only performed when it is needed. This allows TensorFlow to optimize the computation by rearranging the order of operations and reusing intermediate results. TensorFlow also provides a mechanism for automatically

computing gradients of a model, which is necessary for optimization algorithms like stochastic gradient descent. This is achieved using the concept of automatic differentiation, which allows TensorFlow to compute gradients of a function without explicitly computing the derivatives.

Overall, the dataflow system in TensorFlow provides a flexible and efficient way to build and train machine learning models, and it is one of the reasons for TensorFlow's popularity in the machine learning community.

# III. ALGORITHMS

## *A. Logistic Regression:*

Logistic regression[1] is a popular machine learning algorithm used for binary classification problems, where the task is to predict one of two possible outcomes. It models the probability of the binary output as a function of a set of input features. The logistic regression model is essentially a linear model, where the predicted output is obtained by applying a logistic or sigmoid function to the linear combination of input features and their corresponding coefficients. The sigmoid function maps any real-valued number to a value between 0 and 1, which can be interpreted as a probability of belonging to the positive class.

The goal of logistic regression is to find the best set of coefficients that minimize the logistic loss function, which measures the difference between the predicted probabilities and the true labels. This optimization problem is usually solved using gradient descent, a widely used iterative optimization algorithm that updates the coefficients in the direction of the steepest descent of the loss function. In other words, it seeks to find the minimum of the loss function by iteratively adjusting the coefficients until convergence is reached.

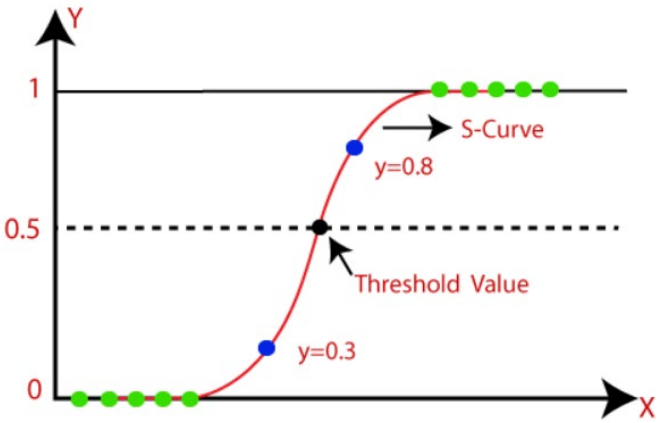


*Figure 3 : Logistic Regression Representation [2]*

The gradient of the loss function with respect to the coefficients is a vector that indicates the direction of the steepest ascent. By taking the negative gradient, we can move in the direction of the steepest descent and minimize the loss function. The gradient is computed using the chain rule of differentiation, by calculating the partial derivatives of the loss function with respect to each coefficient. These partial derivatives represent the contribution of each coefficient to the overall gradient, and they are typically calculated using the data points in a batch or mini batch during training.

In distributed machine learning, the computation of the gradient can be parallelized across multiple machines or nodes, using techniques such as parameter server architectures or data parallelism. This allows us to scale up the training process and handle larger datasets, as well as accelerate the convergence of the optimization algorithm.

### *B. Random Forest*

Random Forest is an ensemble learning algorithm used for classification, regression and other tasks. It is a collection of decision trees where each tree votes to predict the class. The idea is to create a set of decision trees that are diverse and each tree is trained on a random subset of features and samples. In the context of image processing, Random Forest can be used for image classification. In this scenario, the input image is first transformed into a set of features that describe the content of the image. These features could be pixel values, color histograms, texture descriptors, or any other relevant features. Then, a Random Forest model is trained on a set of labeled images, where each image is represented by its feature vector and its corresponding label. [11]

During training, the Random Forest algorithm constructs a set of decision trees, where each tree is trained on a random subset of the features and samples. Each tree predicts a class label, and the final prediction is made by taking a majority vote of the predictions of all the trees.

Once the model is trained, it can be used to classify new images. The image is first transformed into its feature vector, and then the trained Random Forest model is used to predict the class label. The predicted label is the one that receives the most votes from the individual trees in the forest.

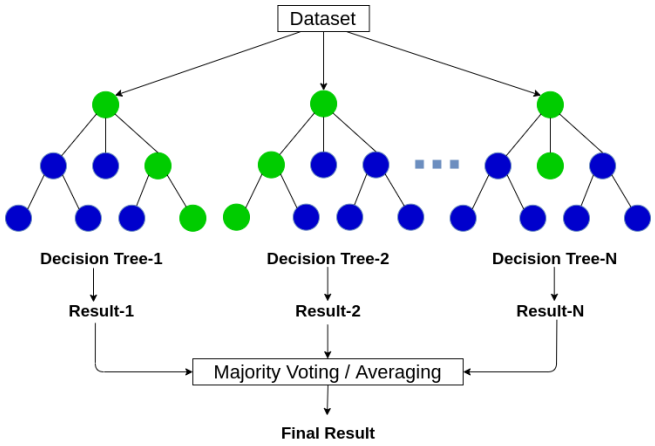


*Figure 4 : Random Forest Algorithm [11]*

Overall, Random Forest is a powerful and flexible algorithm that can be used for image classification and other machine learning tasks. Its ability to handle high-dimensional data, model non-linear relationships, and handle missing data make it a popular choice for many applications.

### *C. Single Layer Neural Network*

A single layer neural network, also known as a single-layer perceptron, is the simplest form of a neural network [5]. It consists of only one layer of artificial neurons (also known as perceptrons) that take input values and produce an output. The input values are first multiplied by a set of weights, and then

summed up to produce an intermediate value. This intermediate value is then passed through an activation function, such as a sigmoid or a *ReLU* function, to produce the final output of the neural network.

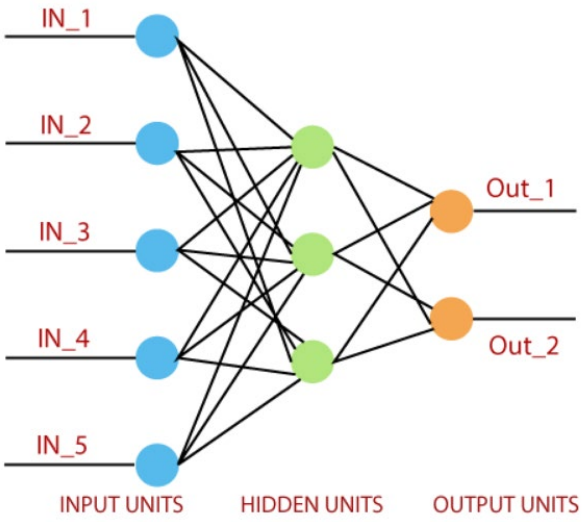


*Figure 5: Single Layer Neural Network [5]*

The weights of the neural network are learned through a process called gradient descent, where the algorithm tries to minimize the error between the predicted output and the actual output. The gradient of the error with respect to the weights is computed using the chain rule of differentiation, and the weights are updated in the opposite direction of the gradient to minimize the error. Single layer neural networks are limited in their expressive power and can only model linearly separable problems. However, they can still be useful in certain situations, such as binary classification tasks.

### *D. Three Layered Neural Network*

A three-layer neural network, also known as a multilayer perceptron (MLP), is a neural network model that consists of an input layer, one or more hidden layers, and an output layer. In this network, information flows from the input layer through the hidden layers to the output layer. Each layer in the network is made up of multiple nodes, also called neurons. The input layer has nodes that represent the input features, while the output layer has nodes that represent the output classes or values. The nodes in the hidden layers use activation functions to transform their inputs into a weighted sum that is passed to the nodes in the next layer.

The main advantage of having multiple hidden layers is that the network can learn more complex representations of the input data. The hidden layers allow the network to identify patterns and relationships that might not be immediately obvious from the raw input data. This is accomplished by using non-linear activation functions, such as the sigmoid or ReLU function, which allow the network to model non-linear relationships between the input features and the output.

Training a three-layer neural network involves adjusting the weights of the connections between the neurons to minimize a loss function. This is typically done using an optimization algorithm such as stochastic gradient descent (SGD) or one of its variants. The gradient of the loss with respect to the

weights is computed using the backpropagation algorithm, which recursively applies the chain rule of calculus to compute the gradient of the loss with respect to each weight in the network.

# IV. Workflow

## *A. Dataset exploration.*

The first step involves exploring and analyzing the data set to understand the underlying patterns, relationships, and distributions in the data. The objective is to identify any missing values, outliers, or other anomalies that could affect the accuracy of the analysis.

## *B. Filtering and pre-processing.*

Once the data set has been analyzed, the next step is to filter and preprocess the data to prepare it for modeling. This step involves data cleaning, feature selection, and normalization. We also store the Data from the CSV to RDDs so that it is in a ready state to apply optimal parallelism.

## *C. Environment Selection.*

Depending on the size of the data set and the complexity of the model, the next step is to select an appropriate environment for model training and evaluation. Popular choices include Apache Spark and TensorFlow. Spark is a distributed computing platform that can process large amounts of data in parallel, while TensorFlow is a popular machine learning library that can be used to build and train a variety of models.

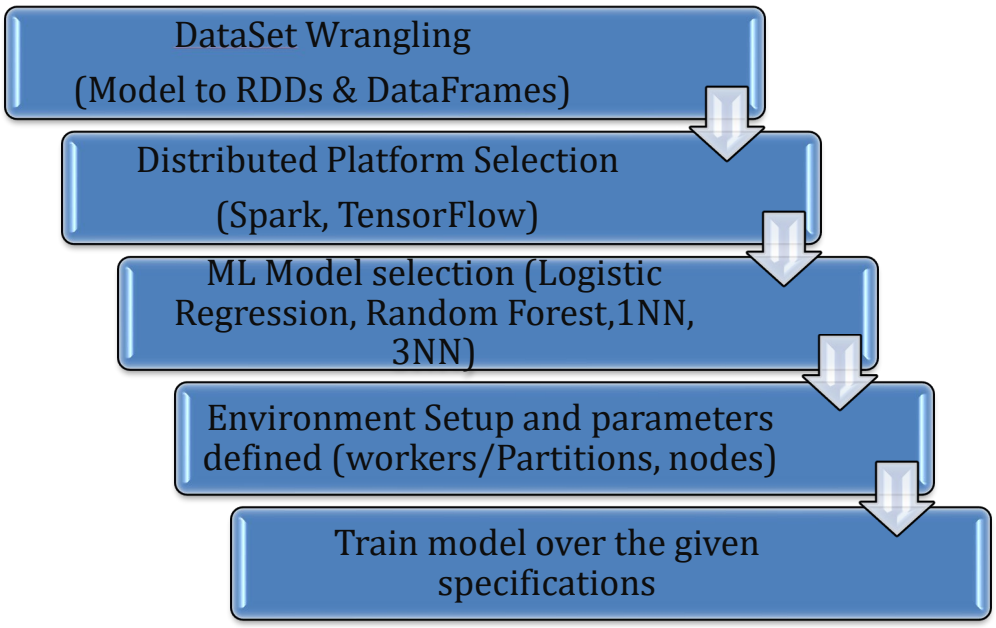


*Figure 6 : Workflow diagram*

### *D. Model Selection.*

Once the environment has been selected, the next step is to select an appropriate machine learning model for the problem at hand. This step involves evaluating different models and selecting the one that performs best on the data set. The algorithms that have been included in this particular experiment include logistic regression, random forest, and k-nearest neighbors (1-NN, 3-NN).

### *E. Experimental Analysis and Results.*

The final step is to conduct experiments to evaluate the performance of the selected model(s) under different conditions. This step involves evaluating the accuracy of the model on the test data set, as well as measuring the CPU and network utilization and the time taken to train the model on a single worker and multi-worker setup. This step provides insights into the scalability and performance of the model under different conditions and helps to identify any bottlenecks or performance issues that need to be addressed.

## V. EXPERIMENTS:

To provide a quantitative evaluation of Spark and TensorFlow, we are evaluating the performance of these two systems with 4 typical machine learning tasks: logistic regression, Random Forest, 1 layered NN and 3 layered NN. All the experiments are conducted on Northeastern University's discovery cloud computing cluster. Each instance contains 4 vCPU powered by Intel Xeon E5- 2676 v3 processor and 16GiB RAM.

### *A. Logistic regression experiments.*

Initially we implemented a two-class logistic regression algorithm on these two platforms. Our Newsgroup dataset contains 6,000 data samples and each of the samples has 80 features. The total size of the dataset is 370MB. Since Spark is suitable for batch data processing, we used full batch gradient descent (batch size =6000) to train the model. The model parameters are stored in Spark's driver (as they fit there) instead of being stored as RDD, Finally in TensorFlow (TF), we implemented synchronous stochastic gradient descent training with varying batch sizes: 100, 500. In these experiments, the cluster of each system contains 10 worker nodes.

We first computed Logistic regression without parallelizing the process. On an experimental analysis to understand the performance parameters. The dataset used in the experimental analysis is the Newsgroup dataset (https://ana.cachopo.org/datasets-for-single-label-text-categorization).

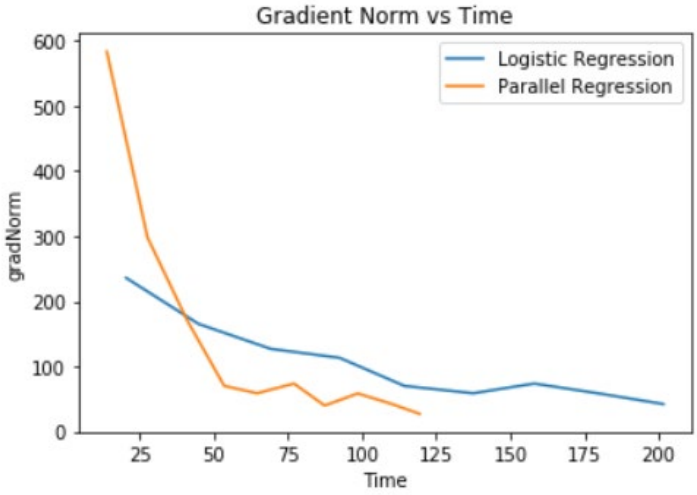


*Figure 7: Logistical Regression, Normal vs Parallelizing*

For a total of N = 10 partitions in Spark, the parallel computation yielded a quicker convergence of the gradient as opposed to normal parallelization.

We also analyzed the total training time taken for 3 different settings (single node Logistic Regression, Parallel Regression (PR) and Parallel Regression (10 partitions)) as follows:

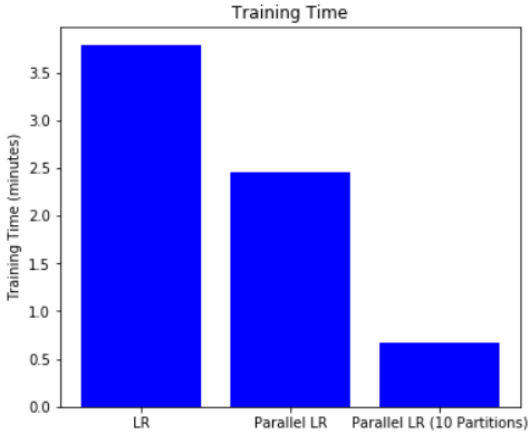


*Figure 8: Training time Comparison for Logistic regression*

We additionally analyzed the CPU utilization using DAG and obtained the following results.

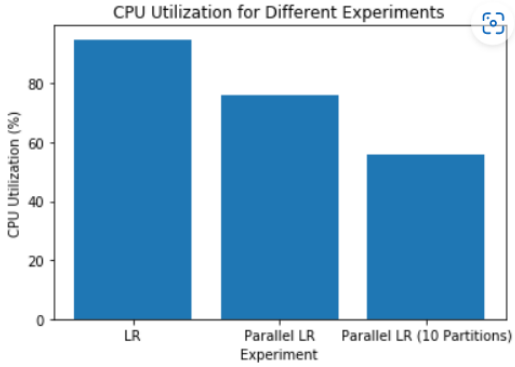


*Figure 8: CPU Utilization for Logistic Regression (Newsgroup dataset)*

Coming to the main experiment, we implemented a Binary class logistic regression algorithm on these two platforms. Our synthetically created dataset contains 70,000 data samples and each of the samples contains 28X28 features. Since Spark is suitable for batch data processing, we used full batch gradient descent (batch size =60000) to train the model. The model parameters are stored in Spark's driver (as they fit there) instead of being stored as RDD. For TensorFlow (TF), we implemented synchronous stochastic gradient descent training with varying batch sizes: 100, 500. In these experiments, the cluster of each system contains (1,3,5) worker nodes and an extra node is needed to serve as the driver in Spark.

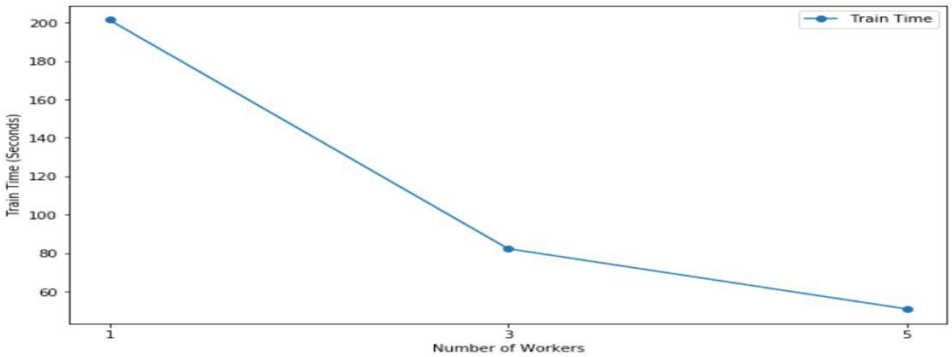


*Figure 9: Workers vs Train time*

Based on the results, it can be observed that increasing the number of workers leads to an increase in CPU utilization and a decrease in training time. However, there is no significant improvement in model accuracy and only a slight increase in memory utilization.

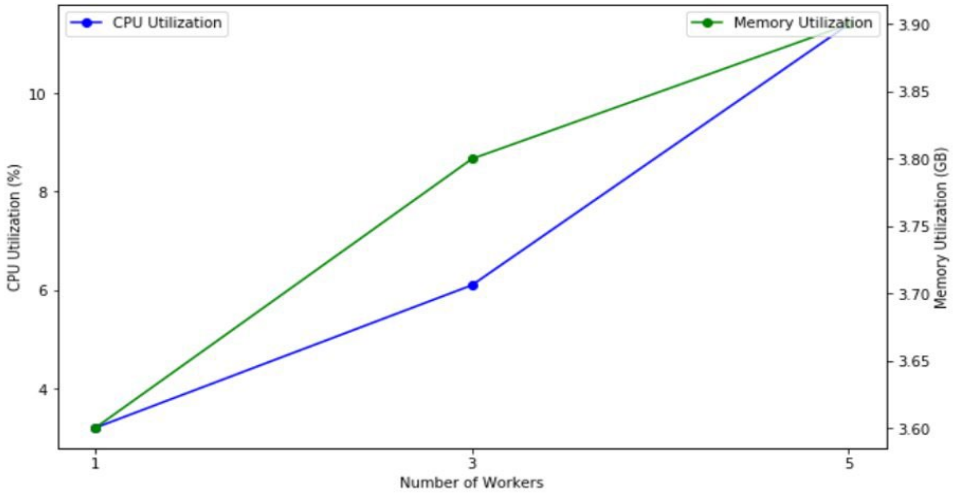


*Figure 10: Resource Utilization*

| Number of Workers | CPU Util % | Train Time (seconds) | Model Accuracy | Memory Utilization |
|---|---|---|---|---|
| Default | 3.2 | 201.393 | 0.9199 | 3.6 |

| | | | | |
|---|---|---|---|---|
| 3 | 6.1 | 82.334 | 0.9199 | 3.8 |
| 5 | 11.4 | 50.982 | 0.9199 | 3.9 |

*Figure 11 : Logistic Regression results*

The default configuration with 3 workers has a CPU utilization of 3.2% and a training time of 201.393 seconds, while the model accuracy is 0.9199 and memory utilization is 3.6. Increasing the number of workers to 3 and 5 leads to higher CPU utilization of 6.1% and 11.4% respectively, and decreased training time of 82.334 seconds and 50.982 seconds respectively. However, there is no significant improvement in model accuracy, which remains at 0.9199, and only a slight increase in memory utilization, which is 3.8% and 3.9% respectively.

### *B. Random Forest Experiments.*

Similarly, we conducted our experiments on evaluating Random Forest by varying (1,3,5) the number of worker nodes in Spark.

| Number of Workers | CPU Util % | Train Time (seconds) | Model Accuracy | Memory Util % |
|---|---|---|---|---|
| Default | 9.2 | 89.602 | 0.946 | 5.1 |
| 3 | 14.7 | 54.620 | 0.9453 | 5.2 |
| 5 | 16.0 | 47.610 | 0.943 | 5.2 |

*Figure 12: Random Forest results*

When comparing these results with the results for logistic regression, we can see that Random Forest has higher CPU utilization and faster training time, but slightly lower accuracy. On the other hand, logistic regression has lower CPU utilization and slower training time, but slightly higher accuracy. Memory utilization is also similar between the two algorithms. Although in practice, logistic regression is generally considered to be fastest in terms of the two, we find that usage of RDDs helped in achieving better speed for Random Forest.

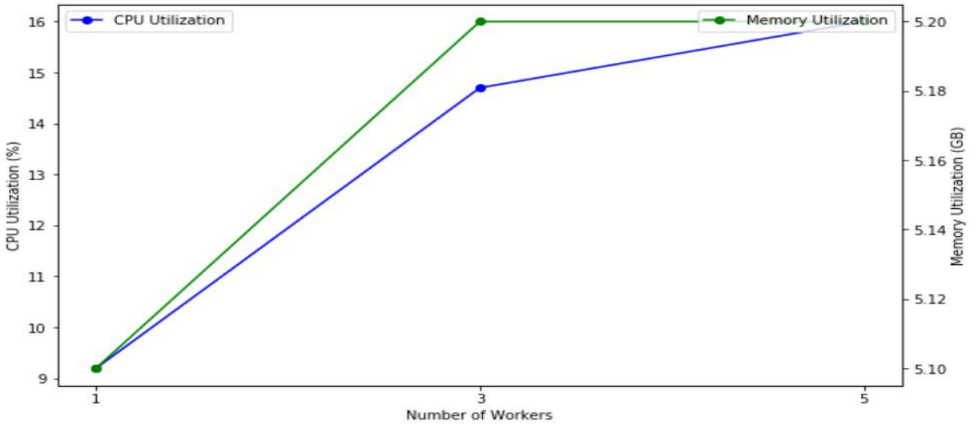


*Figure 13: Resource Utilization*

To get the results and utilization, we utilized the DAG scheduler for Spark along with the *psutil* tool. The results were pretty healthy and gave us a clear overview of our conducted simulation.

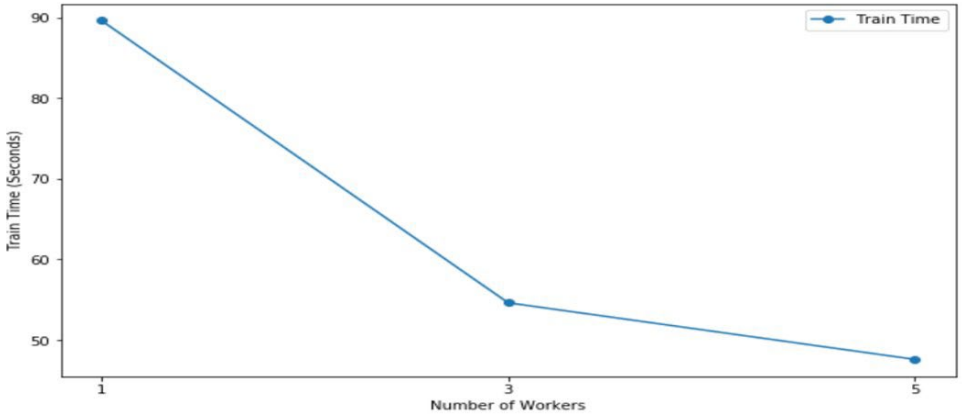


*Figure 14: Workers vs train time*

C. 1-NN and 3-NN Experiments in Spark.

Finally, we conducted our experiments on 1-NN and 3-NN by varying (1,3,5) the number of worker nodes in Spark and in TensorFlow via asynchronous training to evaluate and compare how larger models perform on Spark and TensorFlow which is primarily designed for Larger ML model training.

1-NN (Spark)

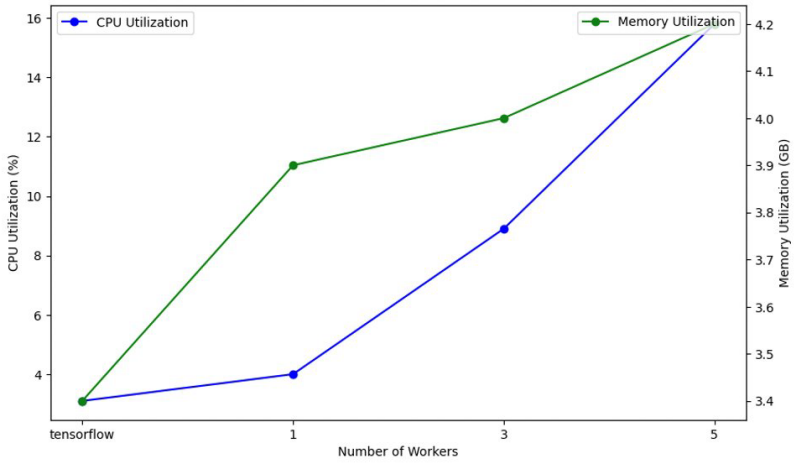


*Figure 15: 1-NN Spark*

| Number of Workers | CPU Util % | Train Time (seconds) | Model Accuracy | Memory Util % |
|---|---|---|---|---|
| Default | 4 | 242 | 0.926 | 3.9 |
| 3 | 8.9 | 104 | 0.9253 | 4 |
| 5 | 15.8 | 87 | 0.92 | 4.2 |

*Figure 16 : 1-NN Spark results*

We can see from Fig 16. that increasing the number of workers decreases the training time for 1-NN in Spark. However, there is a trade-off between training time and model accuracy. While increasing the number of workers from default to 3 reduces the training time by over 50%, the model accuracy remains

almost the same. But increasing the number of workers further to 5, we can see a decrease in model accuracy by 0.006 compared to the default setting. Additionally, as the number of workers increases, the CPU utilization and memory utilization also increase.

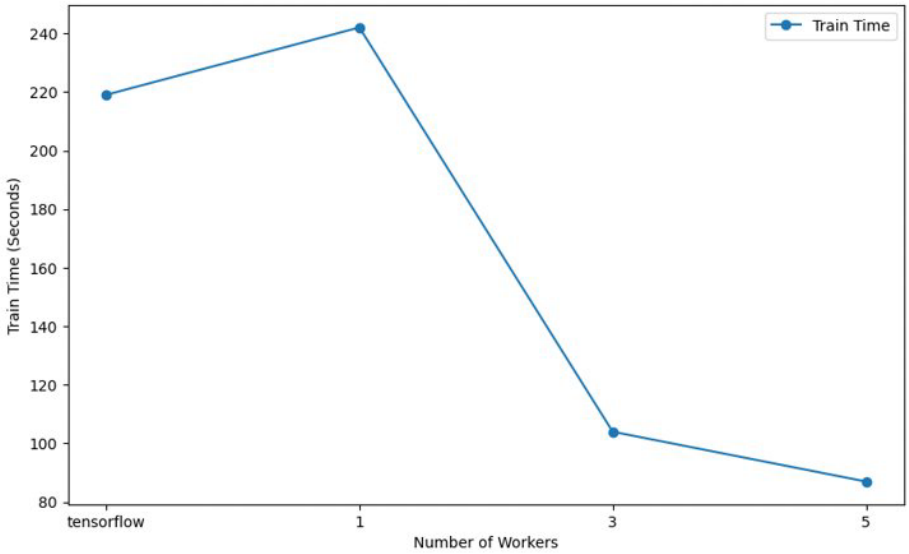


*Figure 17: Worker vs Train time(s)*

3-NN (Spark):

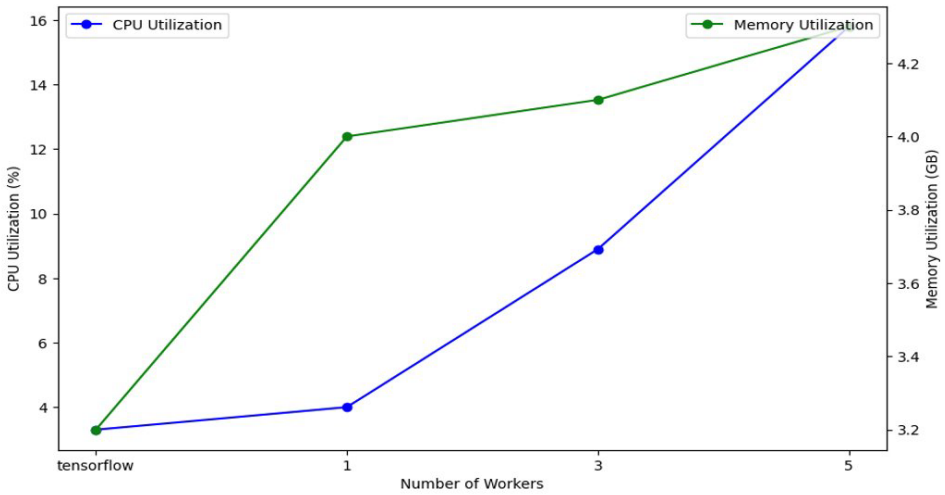


*Figure 18: Resource Utilization in Spark*

| Number of Workers | CPU Util % | Train Time (seconds) | Model Accuracy | Memory Util % |
|---|---|---|---|---|
| Default | 4.1 | 391 | 0.91 | 4.0 |
| 3 | 10.7 | 144 | 0.91 | 4.1 |
| 5 | 16.2 | 102 | 0.912 | 4.3 |
| TensorFlow | 3.3 | 424 | 0.9142 | 3.2 |

*Figure 19: 3-NN results Spark*

From Fig 19. We conducted experiments on the 3-NN algorithm in Spark with different numbers of workers. The results show that increasing the number of workers led to an increase in CPU utilization and a decrease in training time. However, the accuracy of the model remained the same across all experiments, with a value of around 0.91. Memory utilization also increased slightly with an increase in the number of workers.

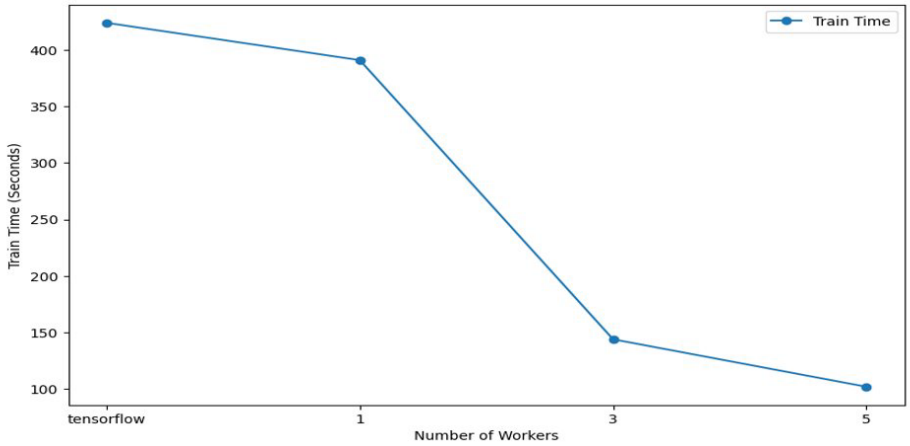


*Figure 20: Worker vs Train time 3-NN*

### *D.* 1-NN and 3-NN Experiments *in TensorFlow*

Apart from Spark implementation, we also decided to setup and experiment on TensorFlow for 1-NN and 3-NN algorithms, all our simulations were conducted on the MNIST dataset. After a set of pre-processing and Data wrangling, we have the ML pipeline ready for implementation.

Next, we implemented the 1NN and 3NN algorithms using TensorFlow's KNN module. For the single worker implementation, we used a single CPU to perform the calculations. For the multi-worker implementation, we used multiple CPUs and distributed the workload using TensorFlow's distributed computing capabilities.

1-NN (TensorFlow):

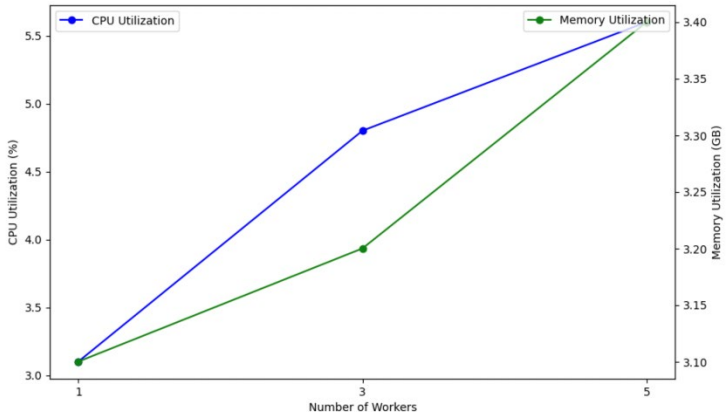


*Figure 21: 1-NN Resource Utilization with TensorFlow*

| Number of Workers | CPU Util % | Train Time (seconds) | Model Accuracy | Memory Util % |
|---|---|---|---|---|
| Default | 3.1 | 219 | 0.926 | 3.1 |
| 3 | 4.8 | 89 | 0.9253 | 3.2 |
| 5 | 5.5 | 42 | 0.923 | 3.4 |

*Figure 22: 1-NN TensorFlow results*

For TensorFlow 1-NN, as the number of workers increased from default to 5, there was a decrease in CPU utilization, train time, and model accuracy. However, there was an increase in memory utilization.

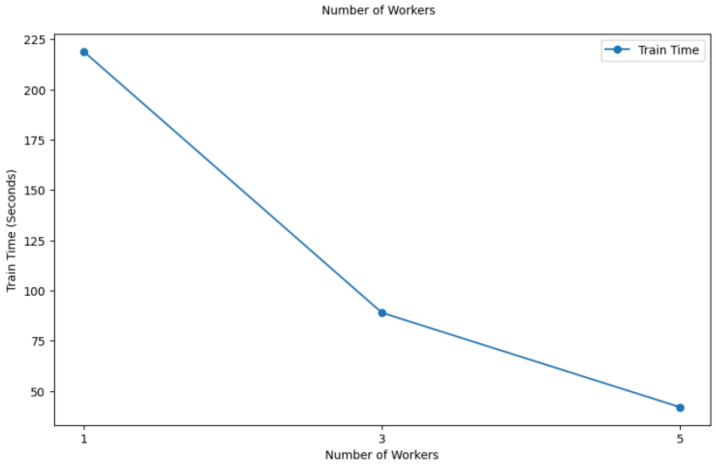


*Figure 23: Workers vs train time*

3-NN (TensorFlow):

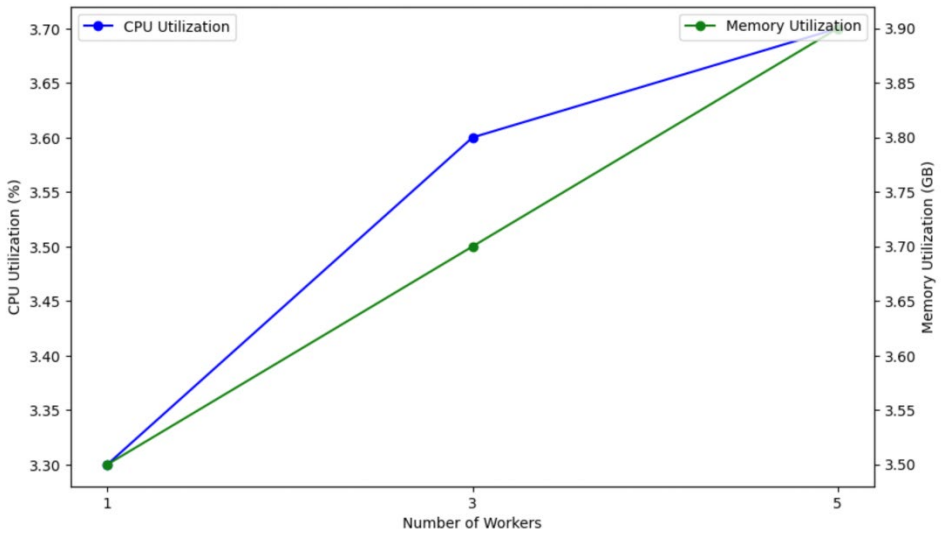


*Figure 24: 3-NN TensorFlow resource Utilization*

| Number of Workers | CPU Util % | Train Time (seconds) | Model Accuracy | Memory Util % |
|---|---|---|---|---|
| Default | 3.3 | 424 | 0.91 | 3.5 |
| 3 | 3.6 | 121 | 0.91 | 3.7 |
| 5 | 3.7 | 22 | 0.913 | 3.9 |

*Figure 25: 3-NN TensorFlow results*

For TensorFlow 3-NN, there was not much change in CPU utilization, train time, and model accuracy as the number of workers increased from default to 5. However, there was a slight increase in memory utilization. It is also observed that the train time is significantly less compared to 1-NN in TensorFlow.

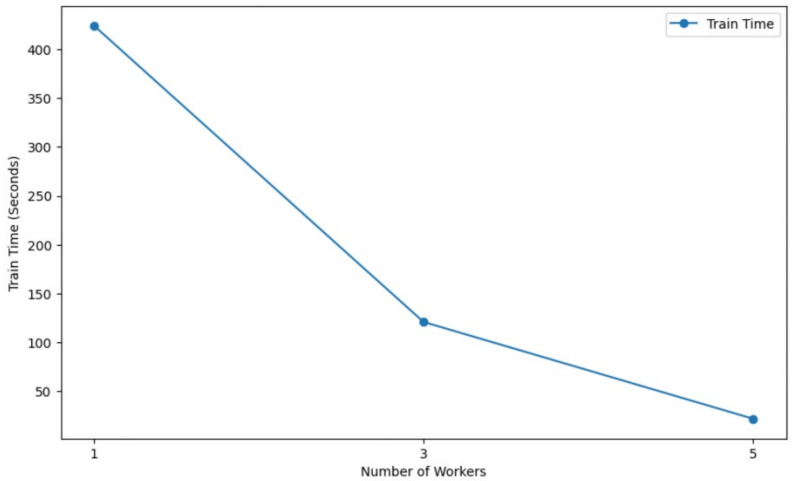


*Figure 26: Workers vs Train time for 3-NN*

| Framework | Model | Best Accuracy | Best Time Taken(s) | Best Memory Utilization | Best CPU Utilization |
|---|---|---|---|---|---|

| | | | | | |
|---|---|---|---|---|---|
| **Spark** | Logistic Regression | 0.9199 | 50.982 (5 workers) | 3.9% | 11.4% |
| | Random Forest | 0.946 | 47.610 (5 workers) | 5.2% | 16% |
| | 1-NN | 0.926 | 87 (5 workers) | 4.2% | 15.8% |
| | 3-NN | 0.912 | 144 (3 workers) | 4.1% | 10.7% |
| **TensorFlow** | 1-NN | 0.926 | 42 (5 workers) | 3.4% | 5.5% |
| | 3-NN | 0.913 | 22 (5 workers) | 3.9% | 3.7% |

*Figure 27: Summary Table*

## VI. CONCLUSION AND SCOPE

From the above presented figures and table 27, we infer that as we increase the complexity of the model, the overall CPU utilization increased significantly whereas the memory utilization has a very slight increase. Finally, as we add more workers, we become increasingly concerned about how much the train time has shrunk. Finally, when we compared the 1-NN and 3-NN implementations in Spark with those of TensorFlow, we found that TensorFlow utilized CPU and memory more efficiently and had faster convergence times for larger models than Spark. This is because TensorFlow uses mini-batch training, whereas Spark does not support mutable state and fast iterations. These needs are addressed by the parameter server model like TensorFlow, which has been widely embraced by platforms for machine learning and deep learning. In order to accommodate the parameter-server architecture, more sophisticated dataflow systems are built that permit cyclic execution graphs with mutable states.

## VII. CONTRIBUTIONS

The simulation and performance evaluation experiment were conducted by Aditya and Bhanu together. Both Aditya and Bhanu were involved in performing EDA of the MSNIT dataset to understand and categorize the features. Aditya was extensively involved in conducting experiments with Logistic Regression, Random Forest and 1-NN in Spark. Apart from clearly understanding and setting up the environment, he also fine tuned the hyperparameters using 4-K Fold cross validation to produce optimal accuracy results for the above Machine Learning Algorithms. Bhanu was responsible for setting up and running the experiments on TensorFlow. He was able to solve the issue with multiple nodes on the Discovery cluster to run and simulate the tensors.

## VIII. REFERENCES


1. Kuo Zhang, Salem Alqahtani, Murat Demirbas, “A Comparison of distributed Machine Learning Platforms”, ICCN 2017

2. https://data-flair.training/blogs/dag-in-apache-spark/

3. Jooat Verbraekan, Matthijs Wolting,Jonathan Katzy,Jeroen Kloppenburg, "A Survey on Distributed Machine Learning", ACM Comput. Surv., Vol. 53, No. 2, Article 30, Publication date: March 2020.

4. https://tsmatz.wordpress.com/2019/09/19/sparkmeasure-metrics-in-apache-spark/

5. D. E. Culler, "Dataflow architectures." DTIC Document, Tech. Rep., 1986.

6. M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in ACM SIGOPS Operating Systems Review, vol. 41, no. 3. ACM, 2007, pp. 59–72.

7. Léon Bottou. 2010. Large-scale machine learning with stochastic gradient descent. In Proceedings of the COMPSTAT'2010. Springer, 177–186.

8. K. Canini, T. Chandra, E. Ie, J. McFadden, K. Goldman, M. Gunter, J. Harmsen, K. LeFevre, D. Lepikhin, T. L. Llinares, et al. 2012. Sibyl: A system for large scale supervised machine learning. Tech. Talk 1 (2012),

9. https://www.tensorflow.org

10. https://www.analyticsvidhya.com/blog/2020/11/introduction-to-spark-mllib-for-big-data-and-machine-learning/

11. https://www.section.io/engineering-education/introduction-to-random-forest-in-machine-learning/