Date of publication December 6th, 2023, date of current version December 6th, 2023.

*Digital Object Identifier 10.1109/ACCESS.2017.DOI*

# Comparison of Established Parallelization Frameworks for Training Various ML Models on ECG Data

**TAYLOR HARTMAN[1], KENDRA GIVENS[1], SCOTT HARTSELL [1], AND KHEM POUDEL.[1]**
[1]Middle Tennessee State University, Murfreesboro, TN 37132

This work is a part of the course CSCI 4330/6330 taught by Dr. Khem Poudel at Middle Tennessee State University

**ABSTRACT** The emergence of Big Data in machine learning has led to an increase in demand for the model(s) with the highest accuracies. However, training multiple algorithms in serial to find the most accurate model is computationally expensive and highly inefficient. Training these algorithms in parallel provides a way to achieve high computational speeds to optimize the comparison process in relation to run-time for cloud applications. Python Threads and Message Passing Interface are standard parallelization frameworks for Python that we use to compare the speed of training some of Apache Spark's Machine Learning library's algorithms logistic regression, Random Forest Decision Trees, and support vector classifier using preprocessed electrocardiogram classification data. We train the three models in serial and then in parallel using each parallelization framework, then compare them to find the one has the highest speed in training. Message Passing Interface performed the fastest in training all three models with Random Forest Decision Trees being the fastest to train and retaining the highest average accuracy. The experiment we conducted has allowed us to find the fastest parallelization framework that can be utilized for cloud computing environments.

**INDEX TERMS** Apache Spark, Big Data, Cloud Computing, ECG, Electrocardiogram, Logistic Regression, Machine Learning, MLlib, MPI, Multi-Processing, Parallel Processing, Python Threads, Random Forest Decision Trees, Support Vector Classifier

## I. INTRODUCTION

ADVANCEMENT in machine learning has increased substantially [1] over the past few years. The introduction of Big Data in this process has increased the potential for higher accuracy in their predictions [2] but also increased the computational time required to train and test effectively. [3] Comparing multiple models on a single dataset to find the one yielding the highest accuracy is a time and computationally intensive endeavour. Training one model at a time and then adjusting hyperparameters to tune the model is too expensive in real-time with large datasets. This lead to the logical solution of programming models to train in parallel. Training in parallel can be highly expensive both in the realms of hardware and in software design, but makes sense in large scale business solutions. Current hardware designs show an exponential increase in computational power as more cores are added to CPU units and highlights the need and ability to handle more simultaneous executions. Machine learning framework developers have improved the issue of run-time complexity by incorporating built-in library model parallelization under the hood.

With many big data machine learning solutions migrating to cloud computing, planning for the billing of these services when training large models with exponentially larger data sets becomes a necessity. Given current industry standards of billing cloud clusters, we can make these three assumptions to billing:

1) The more time you use.
2) The more virtual machines you use (horizontal expansion).
3) The more computational power (cores, GPU's, memory, etc.) you use (vertical expansion) in a cloud system.
4) The more you will be billed for using said system.

We assume the inverse of this is also true. Therefore, we present the problem of being able to train, test, and validate multiple machine learning models using the least amount of time possible and with the least amount of vertical and/or

horizontal expansion possible in order to save the potential user of cloud based systems the most amount of money possible.

For our experiment, we decided to use three machine learning models from Apache Spark's Machine Learning library for our experiment: Logistic Regression, Support Vector Classifier, and Random Forest Decision Trees. Our goal is to be able to train, test, and validate all three models simultaneously after instantiating a single Spark session, on a single compute cluster. In order to draw variance in comparisons and define the most effective approach, we will be applying two different standardized parallelization approaches: Python Threads and Message Passing Interface (MPI). Each model will run 100 times, running a total of 600 models between all threads, and then undergo detailed statistical analysis.

## II. BACKGROUND

### A. APACHE SPARK

Apache Spark is self-described as a "Unified engine for large scale data analytics." [4] This framework is built on top of Hadoop but with some internal changes. Spark has many built-in features that make it ideal for big data analytics. There are roughly 80 high-level operators associated with parallel processing as well as SQL, Streaming (data streaming), MLlib (machine learning), and GraphX (graphs) libraries. [5] These features can work together in unison to create a great computing infrastructure for machine learning and big data processing. Spark provides many advantages for developers to build big data applications. There are two important terminologies in Spark: Resilient Distributed Datasets (RDD) and Directed Acyclic Graph (DAG). Those two techniques can make the speed faster up to ten times as compared to Hadoop. Spark uses memory and the MapReduceing process of spark is faster as compare the Hadoop. Spark bench-marking suite improves the optimization of workload configuration. [6] Spark can handle multiple source data with cached support and perform parallel operation using fault tolerance mechanism. [7] The speed and memory performance of Hadoop and Spark significantly depends on the parameter's configurations such as word load, data size, and cluster architecture.

### B. MACHINE LEARNING ALGORITHMS

#### 1) Logistic Regression (LR)

Logistic regression (LR) is a statistical model that is used to predict the probability of a binary outcome based on one or more independent variables. It is a type of generalized linear model, and is often used in machine learning and data mining. LR was first developed by statistician David Cox in 1958. It was originally used in medical research to predict the probability of a patient surviving a certain treatment. However, it has since been applied to a wide range of other problems, such as predicting whether a customer will churn, whether a loan applicant will default on their loan, or whether an email is spam. LR works by fitting a logistic function to the data. The logistic function is a sigmoid curve that ranges from 0 to 1, and it represents the probability of the outcome occurring. The independent variables are used to determine the parameters of the logistic function, and these parameters are then used to predict the probability of the outcome for new data. LR is a powerful tool for predicting binary outcomes. It is relatively simple to implement, and it can be used with a wide range of data types. However, it is important to note that logistic regression is only a statistical model, and it is not always accurate. It is important to carefully evaluate the model before using it to make predictions. Here are some of the advantages and disadvantages of logistic regression:

**Advantages:**

- LR is a simple and easy-to-understand model.
- It can be used with a wide range of data types.
- It is relatively robust to outliers.
- It can be used to predict both positive and negative outcomes.

**Disadvantages:**

- It can be sensitive to the choice of independent variables.
- It can be difficult to interpret the results.

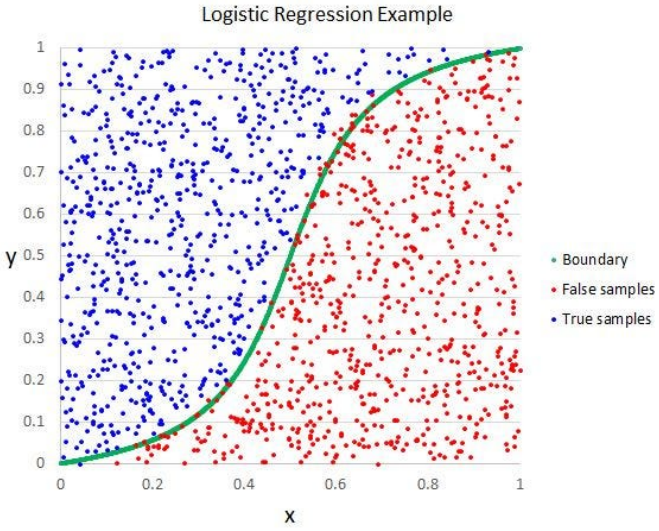



**FIGURE 1.** Visual Representation of how Logistic Regression Classifies

Overall, LR is a powerful tool for predicting binary outcomes. It is relatively simple to implement, and it can be used with a wide range of data types.

#### 2) Support Vector Classifier (SVC)

SVCs are a supervised model used for classification and regression tasks. SVCs separate the data by finding the greatest margin between two support vectors, and defining a hyperplane that lies in the middle. The support vectors span the distance between the closest points of differing classes as shown in 2. In this case, Apache Spark only supports linear classifications, meaning no kernel function is used to project the data into higher dimensions. This means that the solution space is two-dimensional, and our hyperplane is a

one-dimensional vector since the definition of a hyperplane is (solution space - 1) dimensions. The hyperplane is found by randomly guessing and readjusting based off of calculations up to a specified maximum number of tries to maximize the margin between the support vectors. A regularization parameter can be specified to improve the probability of correct classifications, with the trade-off being an increased chance of overfitting.

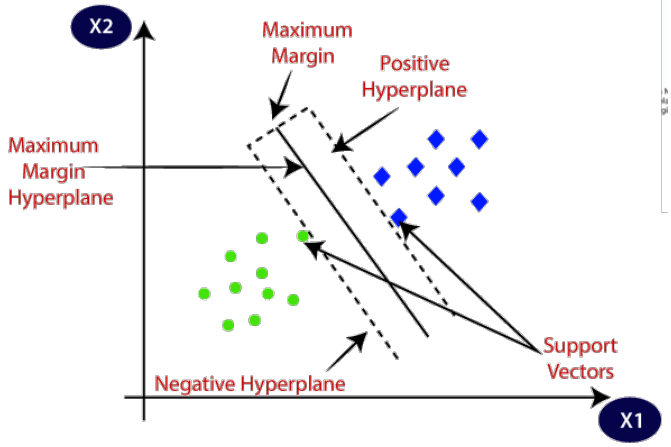



**FIGURE 2.** Example of a Support Vector Classifier

#### 3) Random Forest Decision Trees Classifier (RFT)

The Random Forest (RFT) algorithm utilizes decision trees in order to reach a classification output. First, the presented dataset is "boot-strapped." This means a unique partition is created for use by each decision tree, where samples are randomly selected and duplicate samples are allowed, see figure 3.

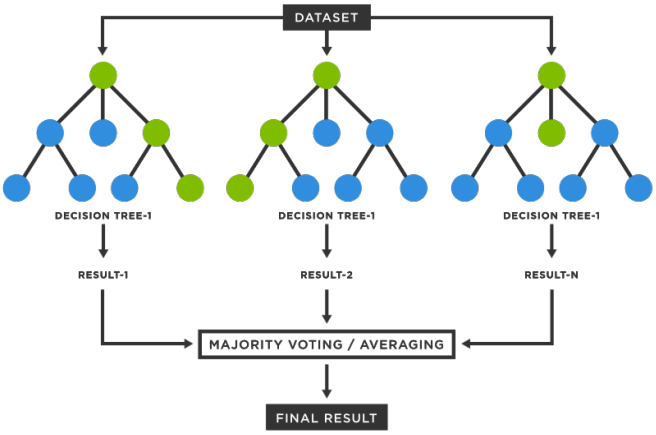



**FIGURE 3.** Example of Random Forest Decision Trees

This spreads data across the forest so that each tree has a unique, limited view of the given problem. A specified number of features from the dataset is then randomly selected for each tree in order to create decision nodes. The recommended number to start with is the square root of total features. Decision nodes are then made based off of the Gini Impurity/Entropy of the feature set, and the Information Gain statistics function. Information Gain is used to determine how much uncertainty can be reduced about a decision made on a feature with relation to another feature. Entropy is calculated for Y, and the entropy of Y given X is subtracted to output the change in uncertainty.

$$IG(Y, X) = E(Y) - E(Y|X)$$

Decision nodes are created up to the maximum depth of the tree as specified by the user, and as the tree is traversed uncertainty is reduced until a decision is made. Trees each output an individual classification of each sample, and "vote" at the end through totaling the count of all classifications. The final output of the Random Forest algorithm is the classification receiving the majority vote. The strategies implemented by this algorithm build a strong foundation for generalization by ensuring the forest is diversified in terms of data view and unique decisions are made locally within each tree.

**Advantages:**

- Control over wide or narrow views of data and features
- Efficient run times

**Disadvantages:**

- Results differ each run due to nature of randomness
- Not all of your data guaranteed to be considered

### C. PARALLELIZATION FRAMEWORKS

#### 1) Python Threads

Python threads are a fundamental concept in concurrent programming, allowing multiple tasks to be executed simultaneously within a single Python process. They provide a way to divide a complex task into smaller, independent parts that can be executed concurrently, resulting in improved performance and scalability. [8]

The concept of threads dates back to the early days of computing, where operating systems introduced the ability to run multiple processes simultaneously. However, processes are heavyweight entities that require their own memory space and resources, making them unsuitable for tasks that can be executed independently but share the same resources.

Python threads address this limitation by providing a lightweight mechanism for concurrent execution. Threads share the same memory space and resources as the main process, allowing them to communicate and cooperate efficiently. They are created using the threading module in Python, which provides a set of functions and classes for managing and synchronizing threads.

Python threads have been instrumental in advancing the field of concurrent programming, enabling developers to write complex and efficient applications that can handle multiple tasks simultaneously. They are widely used in various domains, including web development, data processing,

and scientific computing, where concurrency is essential for achieving high performance and scalability.

#### 2) Message Passage Interface (MPI)

The MPI is a standardized and portable message-passing system designed for parallel programming. It was developed in the early 1990s as a way to enable parallel computing on distributed systems, such as clusters and supercomputers. MPI is a library of functions that programmers can call from C, C++, Fortran, or Python code to write parallel programs. [9]

MPI defines a model of parallel computing where each parallel process has its own local memory, and data must be explicitly shared by passing messages between processes. This is in contrast to shared-memory parallel programming, where processes can access each other's local memory directly. [10]

MPI provides a wide range of functionality for parallel programming, including:

- Communication primitives for sending and receiving messages between processes.
- Process management functions for creating, destroying, and joining processes.
- Synchronization primitives for coordinating the execution of parallel processes.
- Topology functions for describing the geometric arrangement of processes.
- I/O functions for performing input and output operations in parallel.

## III. METHODS

### A. SYSTEM OVERVIEW

Our system consists of three Python files with the same general control flow and execution style. Each program begins with the creation of an Apache Spark session, which allows us to use the machine learning library functions and data frames of the required format. We then import our dataset, "ecg.csv". The dataset is first balanced to have samples from both classifications at a 50-50 ratio to create max entropy in the set. We then split out the classification labels using Python splicing, as they are stored in the last column of the dataset. We then normalize the dataset to a new range while preserving the original relationships within the data. Once the processed datasets have been completed, we instantiate threads from one of two industry standards: Python Threads or MPI.

Each of these standards handles memory sharing as well as message passing differently. Each thread will be passed the Spark Session in order to execute one of three machine learning models simultaneously: one thread is responsible for LR, one for SVC, and one for RFT. The library functions include parallelization under the hood, but without our additional parallelization, the models would still run serially. Each thread will return results: training and validation accuracy, and run-time calculated by subtracting markers created at thread instantiation and completion. Training and validation accuracy are calculated by counting prediction labels in accordance with the confusion matrix. The confusion matrix is restricted to binary classification, and counts the number of true positive, true negative, false positive, and false negative predictions. Once all three threads return from their tasks, the results are stored in an output text file.

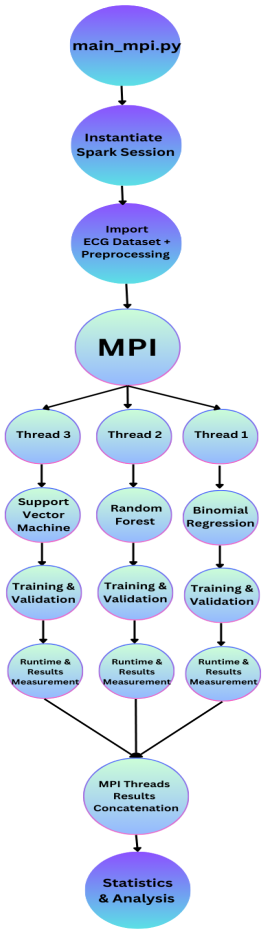



**FIGURE 4.** System design control flow of a single Python file using MPI.

The Python files will be ran 100 times, with an average taken of all results across each file. At the end of the experiment, we will have the average training time for each individual machine learning model, the average total run-time for all of the models, as well as the average accuracy of each individual model. These will be compared to see if there are differences primarily in the run-times of each file and individual model. See Figure 4 for a visual representation of the data flow of the experiment.

### B. STARTING SPARK SESSION

Before doing anything with Apache Spark, first we must install the module and its' dependencies. Once install is completed on your node or cluster, there is one other useful module to install, FindSpark. This module when called will

find and map Apache Spark's install location to your session for ease of starting a session. The code to start a session is rather straight forward. (see figure 5)

**FIGURE 5.** Starting a Spark Session

```
# Creating a sprark session
spark = SparkSession\
        .builder\
        .master("local[1]")\
        .appName("ECG")\
        .getOrCreate()
```

Some methods to take note of, **.master("local[1]")** sets the location of the instance of Spark. [11] The bracketed number one represents the number of CPU cores to be allocated to the process. In these experiment, the instance of Spark is on the local cluster and we chose to use one CPU core. We chose this in an attempt to further standardize the experiment.

### C. DATA PREPROCESSING

The data file [12] we used is a comma delimited (.csv) file. The file does not contain headers, it has one hundred forty-one (141) columns and four thousand nine hundred ninety-eight (4998) rows. Each row is an individual reading with one hundred forty (140) datum points from the ECG; with the one hundred forty-first (141) being the classifier. The classifier is binary in nature (1 or 0) and is present on every column. There are not missing data points within the data set. See figure 6 for a visual representation of the dataset.

| | | | | | | |
|---|---|---|---|---|---|---|
| 1.41096 | 0.788903 | 0.378514 | 0.67799 | 1.278876 | 0.056673 | 1 |
| 0.898527 | 0.860276 | 1.536581 | 1.852605 | 0.618098 | -2.10553 | 1 |
| 1.536242 | 2.03253 | 2.320064 | 1.469294 | 0.821087 | 1.658203 | 1 |
| 1.550276 | 1.137756 | 0.863612 | 0.702984 | 0.890389 | -0.18288 | 1 |
| -1.19575 | -0.8044 | -0.61791 | -1.31842 | -2.13582 | -2.89503 | 1 |
| 1.006196 | 1.092245 | 1.303057 | 1.150097 | 0.790442 | 1.330896 | 1 |
| 1.070206 | 1.759431 | 2.198114 | 2.102171 | 1.22812 | 0.733822 | 1 |
| -3.62041 | -4.21048 | -4.04379 | -3.05181 | -2.80687 | -0.87967 | 0 |
| -3.8812 | -3.92467 | -3.16439 | -2.22837 | -1.83754 | -0.37849 | 0 |
| -3.47217 | -4.28078 | -4.62944 | -3.79419 | -2.85147 | -1.74352 | 0 |
| -2.88488 | -3.53025 | -4.41251 | -3.90333 | -3.56711 | -1.53634 | 0 |
| -2.64039 | -3.00028 | -3.81249 | -3.97336 | -5.22195 | -3.66179 | 0 |

**FIGURE 6.** Sample viewing of the dataset .csv file being used for this experiment

### D. BALANCING DATASET

The initial dataset is imbalanced in reference to the classifier (label) (see Table 1). For better accuracy in regression machine learning models, it is ideal for the classifier ratio to be as close to 50/50 as possible. This is to ensure we are not biasing the model. Introducing biases can cause the model to be weighted more towards the larger inputted label.

To balance the dataset we used Apache Spark to perform our data transformations with the **pyspark.sql** library from within Apache Spark. We initially loaded all of the data into a Spark dataframe, with the schema inferred by Spark, and we renamed the last column to "label". We counted by group the "label" column to get our initial numbers. (see table 1) Then, we separated the data based on the "label" column into their own data frames. That left a dataframe with all the data with a label of 0 and another wit a label of 1. (See figure 7) We then experimented with different ways to randomly remove a specific number of rows from the data frames. We finally ended up using the **randomSplit** function from within Spark with a designated seed of one thousand (1000) that would remove exactly eight hundred forty-four (844) random rows form the data frame with a label of 1 every time the command is ran. Finally, we used the Spark SQL command **.union()** to join the dataset back together.

**TABLE 1.** Imbalanced Classifier in Dataset

| Label | Number | Percentage |
|---|---|---|
| 1 | 2919 | 58.403 |
| 0 | 2079 | 41.596 |

**FIGURE 7.** Code To Balance Dataset

```
df = spark.read.csv("ecg.csv", \
     header=False, inferSchema=True)
df2 = df.filter(df['label'] == '1')
df = df.filter(df['label'] == '0')
df3,df4 = df2.randomSplit([0.30, 0.70],\
                          seed=1000)
df = df.union(df4);
```

Once we have the dataframe "df" with our final data we will use, we deleted all other data frames as they will no longer be used. We did this by using the **del** command on the no longer needed data frames. Additionally, to ensure the memory has been freed up for additional use, we called a garbage collector (gc) to free up that space using **gc.collect()**. The new dataset is now balanced (see figure 2), in one dataframe, and there are no lingering data frames using memory unnecessarily.

**TABLE 2.** Balanced Classifier in Dataset

| Label | Number | Percentage |
|---|---|---|
| 1 | 2975 | 49.952 |
| 0 | 2079 | 50.048 |

### E. SEPARATING FEATURES FROM CLASSIFIER

The next step in our data preprocessing was separating the features from the labels into a separate vector so the model could use the features to train and validate. To do this we need to get the column names of the columns we want as features. This can be an issue with a dataframe that has a lot of column names or a dataframe with no headers, like our dataframe.

We accomplished this by using the **.columns[]** method of the dataframe class of Apache Spark. This method takes in a list of column indexes as its argument. We used Python's slice method to accomplish this in a more general way. We sliced off all of the columns except the last column because the last column is the label column. Apache Spark auto assigns column names when there is no header used. The general convention is an underscore followed by 'c' and the number of the column, with the number of the column starting at 0. (See below for The feature list was created and set to the variable **features_list**. (See figure 8)

**FIGURE 8.** feature_list

```
['_c0',
 '_c1',
. . . # Other columns here
 '_c138',
 '_c139']
```

We then used the function **VectorAssembler** from **pyspark.ml.feature** to make the vector. This function takes in a list and outputs a vector. (see figure 10) In the arguments, we expressly stated the **features_list** as the input columns we want to make into a vector. We then expressly stated that we wanted the output vector column to have the name **feature_vector**. After the assembler was created and set, we used the **.transform()** method to take in our dataframe named **df** and assemble a vector based on the column names in **df** that mateched the column names in **features_list**. This was then appended to the existing dataframe. The **feature_vector** column is appended at the end of the dataframe's columns.

**FIGURE 9.** feature_list

```
from pyspark.ml.feature \
        import VectorAssembler

feature_vector_assembler = \
        VectorAssembler( \
        inputCols=features_list, \
        outputCol="feature_vector") \
df = feature_vector_assembler.\
        transform(df)
```

### F. NORMALIZING THE DATA

We then wanted to normalize the feature data for better performance and training stability in our ML models. To do this, we will be using **StandardScaler** from Apache Spark's Machine Learning library (MLlib) **pyspark.ml.feature**. The **StandardScaler** has options to standardize based on the **Mean** and the **Standard Deviation**. In our case, we chose to utilize both of these options (see figure 10).

**FIGURE 10.** Scaled Feature Data Vector

```
scaler = StandardScaler(\
    inputCol="feature_vector",\
    outputCol="scaled_feature_vector",\
    withStd=True, withMean=True)

scaler = scaler.fit(df)

df = scaler.transform(df)
```

The function takes in an input vector, in our case it is **feature_vector**. It outputs a vector, which we named **scaled_feature_vector**. The flags **withSTD** and **withMean** have been set to **True** so the function uses those methods of normalization when normalizing the data. This is initialized to a variable, that variable is then fit to the dataframe and set to the same variable, that StandardScaler variable is then transformed on the dataframe, and set to the dataframe. This creates the new **scaled_feature_vector** and appends the new column with scaled feature vectors for each row to the end of the dateframe **df**. See table 3 for the first five features of the first row of data in both their raw values and normalized values.

**TABLE 3.** Raw vs. Normalized Feature Vectors

| Raw | Normalized |
|---|---|
| -0.032245388 | 0.17704348342429316 |
| -0.54504988 | 0.6956519582807159 |
| -0.82233951 | 1.0789257870215128 |
| -1.6050845 | 1.0121485426922339 |
| -1.8057748 | 1.0779257153411645 |

### G. JOINING BACK FEATURES AND LABELS

With the features normalized and together in a vector, it is time to join the features back together with the labels so the ML models can use the data for training, test, and validate. We do this by creating a new dataframe (model_df) and using the **select()** SQL method (see figure 11)

**FIGURE 11.** Join Scaled Features and Labels into New Dataframe

```
model_df = df.select(\
    "scaled_feature_vector",\
    "label")
```

This process is rather straight forward and results in the data in a dataframe that we used to train, test, and validate the ML models. (See figure 12)

### H. SETTING UP FOR TRAINING

The final step in our data preprocessing was to split the now normalized feature vectors and labels into separate training,

**FIGURE 12.** model_df

```
+--------------------+-----+
|scaled_feature_vector|label|
+--------------------+-----+
| [0.17704348342429...|    0|
| [0.20291272241540...|    0|
| [0.38156610598271...|    0|
+--------------------+-----+
```

testing, and validation dataframes. We did this by using the function **.randomSplit()** (See fig 13). We split the data into training_df, test_df, and validation_df in 70%, 20%, and 10% portions. We again used the **seed** argument to establish normalcy between individual runs with the seed being twenty-two (22). The final split numbers can be seen in figure 4.

**FIGURE 13.** Data Split for Training, Testing, and Validation

```
training_df, test_df, validation_df =\
    model_df.randomSplit(\
    [0.7, 0.2, 0.1],\
    seed=22)
```

**TABLE 4.** Percentage Split for Training, Testing, and Validation

| Name | Number | Percentage |
|---|---|---|
| Training | 2886 | 69.475 |
| Test | 826 | 19.884 |
| Validation | 442 | 10.640 |

We now have all of our features separated, in a vector, normalized based of the dataset's mean and standard deviation, and we have that vector of normalized features and the corresponding labels for those feature vectors in a stand lone dataframes which have been split into sections for training, testing, and validation based on the 70/20/10 split method.

### I. TRAINING THE MODELS

The three algorithms were run one hundred (100) times, independently, on a single AMD EPYC 7402 24-Core Processor through Middle Tennessee State University's cluster system and the resultant accuracies of the models and training times were averaged. This includes three different Python scripts containing the three different frameworks.

### J. MPI

MPI proved challenging due to its' unorthodox data flow. It is worth noting that our initial experiments to utilize MPI for model training parallelization were unsuccessful. We learned that the contributing factor to this was utilizing a single process to start a spark session, pre-process the data, and then pass that data structure to other threads for model

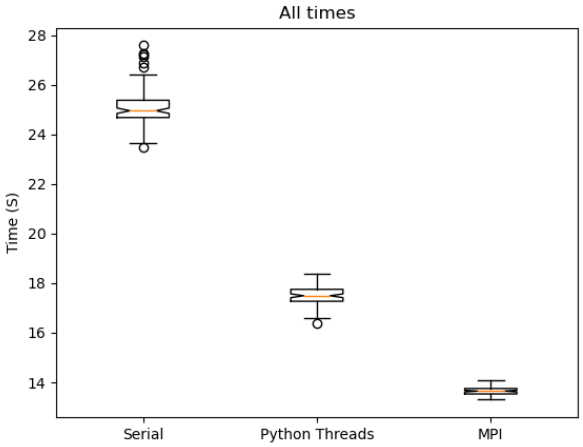



**FIGURE 14.** Sample viewing of the dataset .csv file being used for this experiment

training. This did not work due to threads not inherently sharing data, therefore the threads responsible for training the models, could not see the Spark instance that is required to use Spark's MLlib. To fix this issue, we set all of those within the main so all processes would have access to the data. This fixed the issue and the experiment was conducted.

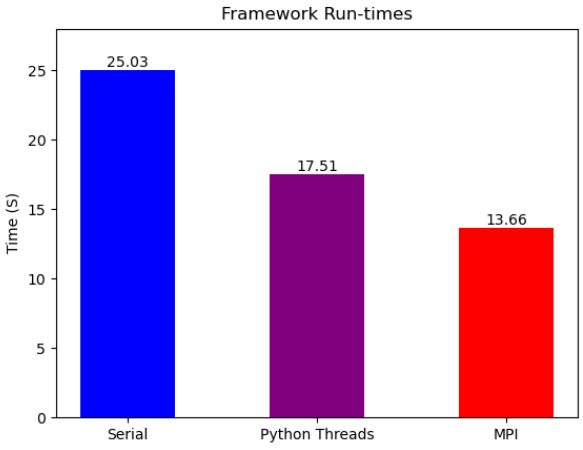



**FIGURE 15.** Framework Run-times.

## IV. RESULTS

### A. PARALLELIZATION

The performance of training the models in parallel did not agree with our initial hypothesis. Initially we had asserted that the Python Threads would perform the fastest as it was less computationally expensive. However, during our research we discovered the presence of a Global Interpreter Lock (GIL) within Python that halted true parallelization. Although the performance of Python Threads was not as

expected due to the GIL, the syntax was much cleaner and easier to comprehend than MPI.

Figure 14 represents a notched box plot of all of the run times for the one hundred (100) individual runs of the frameworks. MPI holds the advantage in the time it took to train all three algorithms. Figure 15 shows the median run-times for the three frameworks.

### B. MACHINE LEARNING MODELS

Apache Spark's Machine Learning Algorithms from MLlib performed very well with their default settings. The standardization of the dataset did improve the models' accuracies on validation data by one percentage point (1%). This is not shown in the results here but it is worth noting that normalization of the dataset did improve accuracy on unseen data.

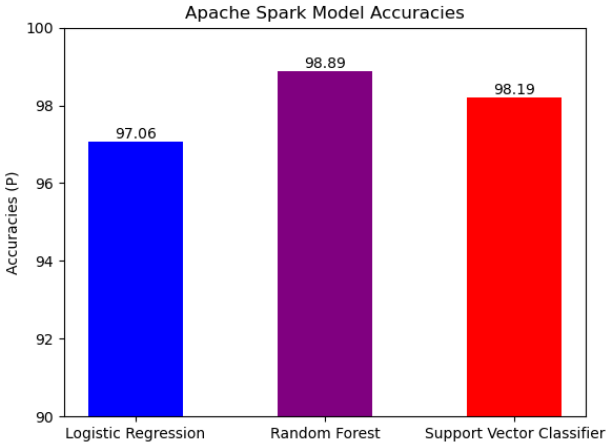



**FIGURE 16.** Apache Spark Machine Learning Algorithm Accuracies.

As shown in figure 18, we can see that RFT produces the highest average accuracy across the one hundred (100) independent runs. It is worthwhile to note that the LR and SVC models accuracies did not change when training the models due to the seed placed when splitting the data into sections during data preprocessing. We hypothesize that forced the model use the same rows of data for training, testing, and validation every run. This, however, is not the case for the RF algorithm due to the nature of the algorithm itself. RF's random sampling is the reason that the accuracy of the RF model was different every time compared to the LR and SVC models. Figure 18 shows the confusion matrices for the three models tested. This shows minimal variance in the models. However, the model with the fastest training speed and highest accuracy was Random Forest. (see figure 16 and 17).

## V. DISCUSSION

If the choice arises to use serial code, Python Threads, or MPI to train multiple Apache Spark machine learning algorithms, the choice is clear if speed is the highest concern: MPI. MPI

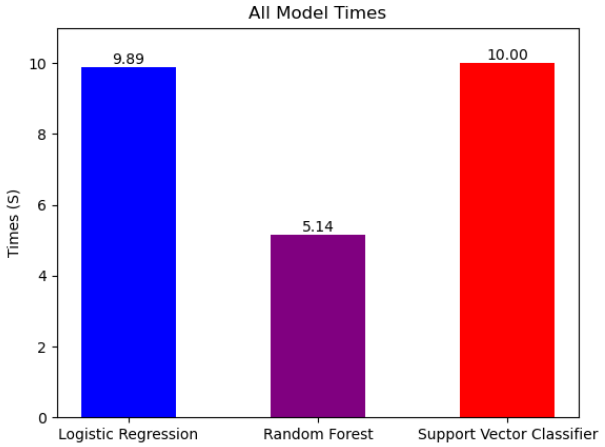



**FIGURE 17.** Apache Spark Machine Learning Algorithm Run-times.

trained the three models 22% faster than Python Threads and in almost half the time of the serial code. However, if you are looking for a slight time improvement without sacrificing easy code readability, Python Threads would be an alternate choice. In the end, MPI ran as fast as the slowest model was able to train: the Support Vector Classifier. Future work would consist of utilizing a larger dataset to see if the results change, comparing more machine learning algorithms and frameworks, compare the memory usage of each framework while parallelizing training, and compare Python Threads without GIL to the other frameworks.

## REFERENCES

[1] M. I. Jordan and T. M. Mitchell, "Machine learning: Trends, perspectives, and prospects," Science, vol. 349, no. 6245, pp. 255–260, Jul. 2015, publisher: American Association for the Advancement of Science. [Online]. Available: https://www.science.org/doi/10.1126/science.aaa8415

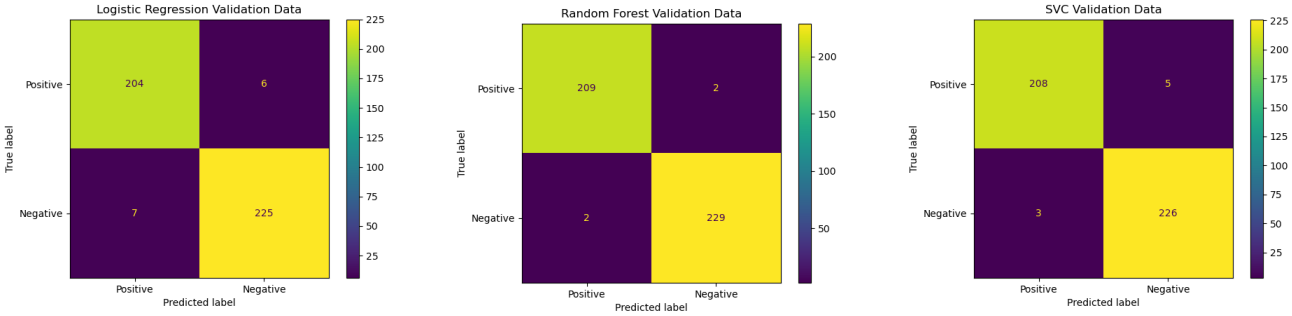
[2] W. Raghupathi and V. Raghupathi, "Big data analytics in healthcare: promise and potential," Health Information Science and Systems, vol. 2, p. 3, Feb. 2014. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4341817/

[3] K. Batko and A. Ślęzak, "The use of big data analytics in healthcare," vol. 9, no. 1, p. 3. [Online]. Available: https://doi.org/10.1186/s40537-021-00553-4

[4] Apache spark™ - unified engine for large-scale data analytics. [Online]. Available: https://spark.apache.org/

[5] S. Alotaibi, R. Mehmood, I. Katib, O. Rana, and A. Albeshri, "Sehaa: A big data analytics tool for healthcare symptoms and diseases detection using twitter, apache spark, and machine learning," vol. 10, no. 4, p. 1398, number: 4 Publisher: Multidisciplinary Digital Publishing Institute. [Online]. Available: https://www.mdpi.com/2076-3417/10/4/1398

[6] N. Ahmed, A. L. C. Barczak, T. Susnjak, and




Logistic Regression | Random Forest Decision Trees | Support Vector Classifier

**FIGURE 18.** Apache Spark Machine Learning Model's Confusion Matrices

M. A. Rashid, "A comprehensive performance analysis of apache hadoop and apache spark for large scale data sets using HiBench," vol. 7, no. 1, p. 110. [Online]. Available: https://doi.org/10.1186/s40537-020-00388-5

[7] H. Ahmadvand, M. Goudarzi, and F. Foroutan, "Gapprox: using gallup approach for approximation in big data processing," vol. 6, no. 1, p. 20. [Online]. Available: https://doi.org/10.1186/s40537-019-0185-4

[8] "threading — Thread-based parallelism." [Online]. Available: https://docs.python.org/3/library/threading.html

[9] "MPI Forum." [Online]. Available: https://www.mpi-forum.org/

[10] "MPI for Python — MPI for Python 3.1.5 documentation." [Online]. Available: https://mpi4py.readthedocs.io/en/stable/

[11] "Getting Started - Spark 3.5.0 Documentation." [Online]. Available: https://spark.apache.org/docs/latest/sql-getting-started.html

[12] ECG dataset. [Online]. Available: https://www.kaggle.com/datasets/devavratatripathy/ecg-dataset