Thesis Dissertation

Leveraging Apache Spark and Fogify Emulation Framework for Traffic Prediction: A Machine Learning Approach

Marios Vasiliou, Undergraduate student, Computer Science

UNIVERSITY OF CYPRUS

COMPUTER SCIENCE DEPARTMENT

May 2023

UNIVERSITY OF CYPRYS

COMPUTER SCIENCE DEPARTMENT

Leveraging Apache Spark and Fogify Emulation Framework for Traffic Prediction: A Machine Learning Approach

Marios Vasiliou

Supervisor

Dr. George Pallis

Thesis submitted in partial fulfilment of the requirements for the award of degree of Bachelor in Computer Science at University of Cyprus

May 2023

Acknowledgements

I would like to thank my supervisor professor Dr. George Pallis, for giving me the opportunity to work with him. Equally, I am grateful to Dr. Moysis Symeonidis for his continuous support and help.

I would also like to thank my family for all the financial, mental, and emotional support throughout my studies. A special thanks to my friends and girlfriend who have been with me in this journey. Their companionship, understanding, and unwavering support have been fundamental in helping me reach this point in my academic pursuits.

Abstract

This thesis presents an integrated approach to the development and performance evaluation of a traffic volume prediction application using the SparkEdgeEmu Emulation System, built on top of the Fogify Emulation Framework. The application aims to forecast traffic volume, a crucial factor in managing and planning urban traffic flow. It processes traffic volume data within a Spark standalone cluster on Docker, utilizing machine learning algorithms including Multilayer Perceptron Classifier, Gradient-Boosted Trees, and Random Forest Classifier. These algorithms were selected due to their effectiveness in handling large-scale data, their capacity to model complex nonlinear relationships, and their robustness against overfitting. The application is subsequently deployed within SparkEdgeEmu, leveraging the Fogify Emulation Framework to conduct thorough experimentation and evaluation in a realistic edge computing environment. Performance metrics and resource utilization of each algorithm are analyzed in detail. This research underlines the potential of integrating advanced machine learning techniques with the SparkEdgeEmu and Fogify Emulation Frameworks for the development and optimization of intelligent transportation systems, addressing real-time traffic management and planning challenges effectively.

Contents

1. Introduction

The advent of Internet of Things (IoT) devices and the corresponding surge in data generation have driven the need for inventive solutions capable of efficient data processing and analytics. Edge computing, often complemented by cloud computing, provides a promising framework for these tasks. However, the deployment and testing of such systems in real-world scenarios can be both expensive and time-consuming. This thesis aims to address this issue.

This project harnesses a combination of technologies - Docker, Spark, and machine learning algorithms - to establish a scalable, real-time, and precise traffic prediction system for smart cities. Docker supplies the infrastructure for building and managing the application, while Spark, a distributed data processing system, serves as the backbone of the data analytics pipeline. For code development and testing, we employ JupyterLab, an interactive development environment.

In addition, we utilize SparkEdgeEmu, an innovative tool that emulates edge computing environments and facilitates the evaluation of Spark analytic jobs without the need for a full-scale edge topology setup. SparkEdgeEmu provides parameterizable templates for edge infrastructures, enabling the creation of realistic, emulated environments. It also provides a unified and interactive programming interface, along with comprehensive metrics from the emulated topology and the executed queries.

After local testing and debugging, the code of the project is deployed to Fogify, an emulation framework for fog and edge computing. With the support of SparkEdgeEmu, Fogify facilitates the emulation of the application's deployment on edge nodes distributed across a smart city, enabling the extraction and analysis of various benchmarks and edge metrics.

The effective management of traffic flow is a critical challenge for modern cities. As the global population grows, so does the need for new – better traffic management strategies. Traffic volume predictors could be the key, providing valuable insights that can guide traffic management decisions. To this end, a traffic volume predictor that utilizes real-time traffic data to anticipate congestion and provide valuable insights for traffic management decisions

is developed. By analyzing traffic data in real time, we can discern patterns and trends, anticipate congestion, and propose solutions to alleviate it.

The overarching goal of the project is to demonstrate the transformative potential of edge computing in evolving urban landscapes into smart cities. It is posited that this approach can deliver essential insights into traffic management, alleviate congestion, and ultimately contribute to a more sustainable urban environment.

2. Related Work

Traffic volume prediction plays a crucial role in traffic management and planning. The accuracy and reliability of these predictions have been significantly enhanced through the utilization of various predictive models.

Among these, traditional models like the Autoregressive Integrated Moving Average (ARIMA) have been extensively employed for time-series forecasting, including traffic volume prediction [1]. ARIMA models are based on the idea that information in past values of the time-series can alone be used to predict the future values. However, these models often assume linear relationships and are usually characterized by constant variance over time, assumptions which may not sufficiently encapsulate the complexities and non-linearities inherent in traffic patterns.

To surmount the limitations of these conventional models, machine learning algorithms have been introduced for traffic volume prediction. Algorithms ranging from decision trees and support vector machines to advanced deep learning techniques like Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) have been employed. Specifically, CNNs, known for their efficacy in image and speech recognition tasks, have been adapted to capture spatial dependencies in traffic data, whereas RNNs, especially Long Short Term Memory (LSTM) networks, have shown considerable promise in handling time-dependent patterns inherent in such datasets.

In line with the advancements in big data processing, the application of Apache Spark for traffic prediction has gained momentum in recent studies, with the focus largely on real-time traffic data streaming and processing using Spark Streaming. However, the integration

of Apache Spark with various machine learning models, for the explicit purpose of traffic volume prediction, remains an under-researched area.

In the quest to create simulations that closely mirror real-world scenarios, emulation frameworks like Fogify and SparkEdgeEmu have emerged as invaluable tools. SparkEdgeEmu, which employs Fogify as its underlying emulation framework, has been particularly promising in deploying Spark applications in edge computing environments, replicating real-world network conditions and constraints.

This project seeks to bridge the existing research gap by employing various machine learning models, including the Multilayer Perceptron Classifier, Gradient-Boosted Trees, and Random Forest Classifier, within the Apache Spark environment for traffic volume prediction. Explicitly, the project utilizes the iterative learning process of the Multilayer Perceptron, the powerful hierarchical nature of Gradient-Boosted Trees, and the robust and adaptable ensemble method of Random Forest Classifier to develop accurate predictive models. Furthermore, the project employs SparkEdgeEmu and Fogify to emulate real-world edge computing conditions, enabling a comprehensive evaluation of the performance and scalability of the developed models under various network conditions and workloads.

3. Background

This section provides a comprehensive understanding of the foundational technologies and concepts that are integral to this thesis. Platforms, frameworks, and concepts such as JupyterLab, Docker, Apache Spark, and Fogify are discussed, along with a variety of machine learning algorithms, edge computing paradigms, and smart cities. These topics contribute significantly to comprehending the context and impetus behind the research conducted in this thesis.

3.1 JupyterLab

JupyterLab is an open-source, web-based interactive development environment (IDE), providing a notebook interface for creating, editing, and executing Jupyter notebooks. JupyterLab was chosen as a key tool in this project due to its significant compatibility with

Apache Spark and Docker. Moreover, it provides a flexible workspace that enables interactive data visualization and integration of multiple languages. JupyterLab's extensive range of extensions further enhances its utility, facilitating tasks such as code versioning and formatting. The capacity to include narrative text, mathematical equations, and rich media within the same document makes JupyterLab an ideal platform for composing reproducible scientific documents and sharing the process of data exploration and analysis.

3.2 Docker and Spark Standalone Cluster on Docker

Docker, an open-source platform, has transformed the process of application development, deployment, and execution through the use of containers. Containers wrap an application and its dependencies into a lightweight and portable unit, ensuring consistent performance across various environments. Owing to its ease of use, scalability, and flexibility, Docker has become indispensable in modern software development. [1]

For the purposes of this project, Docker enables the seamless deployment and orchestration of the traffic volume prediction application across a distributed computing environment.

Key concepts in Docker include:

a. Containers: Containers are lightweight, standalone, and portable units that contain an application and its dependencies. They run in isolation from one another, ensuring that the application behaves consistently across different environments. Containers provide a consistent and reproducible runtime environment, making it easier to develop, test, and deploy applications seamlessly across various stages of the software development lifecycle. [2]
b. Images: Docker images are read-only templates used to create containers. They contain the application code, runtime, system tools, libraries, and settings required to run the application. Images are built from a set of instructions specified in a Dockerfile, which defines the base image, the application code, and other necessary components. Images can be stored in a registry, such as Docker Hub, allowing developers to share and distribute their images easily. [2]

c. Docker Compose: Docker Compose is a tool for orchestrating and running multi-container Docker applications. It uses a YAML file to configure the application's services, networks, and volumes, allowing developers to manage the entire application stack with a single file. Docker Compose simplifies the process of deploying complex applications by enabling developers to define their application's infrastructure and dependencies in a structured and human-readable format. This approach promotes collaboration among team members and streamlines the deployment process, making it more efficient and reliable. [1]

To accommodate the processing requirements of the traffic volume prediction application, a standalone Spark cluster is established within Docker. This setup provides an isolated environment, ensuring that dependencies and configurations for the Spark cluster do not interfere with other system components. This abstraction decouples the application logic from the underlying execution environment, boosting its portability and scalability.

The Docker-based Spark Standalone cluster utilized in this project consists of a Spark master, two Spark workers, and JupyterLab. The cluster is orchestrated using Docker Compose, defining the dependencies and relationships among these components. The use of Docker simplifies the deployment and configuration of this Spark Standalone cluster, allowing for a quick setup of a reliable and consistent platform for the application. Furthermore, Docker facilitates efficient utilization and management of resources such as CPU and memory, ensuring the portability and reproducibility of the Spark Standalone cluster environment. Docker's capabilities thus enable the seamless integration of the Spark Standalone cluster with other project components, such as data ingestion and Spark processing, to efficiently conduct traffic volume prediction.

3.3 Apache Spark

Apache Spark is a distributed computing system designed for processing large-scale data fast and efficient. It offers a unified platform for big data processing, machine learning, graph processing, and stream processing. In this thesis, we are using Spark’s capabilities to implement and evaluate various machine learning algorithms, in a fog computing environment, using the PySpark library.

The main advantages of Apache Spark, which motivated its selection for this project, include:

a. In-Memory Processing: Spark can store intermediate data in memory, which significantly reduces the time it takes to access and process the data. [3] This is particularly useful for machine learning tasks, where iterative algorithms and multiple data passes are common.
b. Scalabiliy: Spark can scale out to handle large volumes of data by distributing the processing workload across multiple nodes in a cluster. This enables the application to handle the large-scale traffic data used, while maintaining good performance.
c. Flexible APIs: Spark offers APIs in various programming languages, including Python. By using PySpark, machine learning algorithms could be easily implemented in a familiar language while still benefiting from Spark's powerful processing capabilities.
d. MLlib: Spark provides MLlib, a built-in machine learning library that includes a wide range of algorithms and utilities. This allowed to easily experiment with different machine learning models and techniques for predicting high traffic conditions. [4]
e. Integration with Cluster Managers: Spark can be deployed on various cluster management systems, such as the standalone cluster, which was used. This enabled the easy deploy in a Fog computing environment. [5]

To the point, Apache Spark’s distributed processing capabilities allows to process the traffic volume data used in this project efficiently. It enables performing feature engineering like extracting new features without performance bottlenecks. This becomes particularly crucial when conducting experiments and benchmarking the performance of various machine learning algorithms. In addition, by using Spark’s distributed processing capabilities, the project could be deployed within the Fogify Emulation Framework, for the examination of its behavior in an edge topology.

3.4 Emulation with Fogify and SparkEdgeEmu

Emulation is a critical aspect of modern computing that allows for the mimicking of specific system behavior, offering a pragmatic means to simulate real-world scenarios without the requirement of physical hardware or extensive resources. In this context, emulation frameworks such as SparkEdgeEmu and Fogify are invaluable for testing, optimizing, and evaluating applications under various conditions and constraints.

SparkEdgeEmu is a tool tailored to aid researchers and practitioners in examining the performance of Spark analytic jobs within edge computing environments. This interactive framework removes the complexity of setting up an edge topology, instead offering parameterizable templates for edge infrastructures and real-time emulated environments with ready-to-use Spark clusters. SparkEdgeEmu also provides a unified and interactive programming interface for executing and submitting queries, along with metrics detailing both the utilization of the underlying emulated topology and the performance of the deployed queries.

On the other hand, Fogify is a specialized emulation framework designed for fog computing environments. It empowers users to define, deploy, and scrutinize fog applications under practical network, compute, and storage conditions. This allows researchers and developers to ensure their applications are robust, scalable, and performant in real-world fog computing environments.

In the context of this project, the traffic volume prediction application is deployed within Fogify. Using SparkEdgeEmu as an intermediary, Fogify emulates the conditions of an edge topology (smart-city scenario), enabling us to evaluate the application's performance under realistic conditions. This interplay between SparkEdgeEmu and Fogify provides a comprehensive toolset for the effective design, testing, and evaluation of fog and edge computing applications, which is a major focus of this thesis.

3.5 Machine Learning Algorithms

Machine learning algorithms are essential for the development of intelligent traffic prediction and management systems. In this thesis, we explore several machine learning algorithms, including Random Forest, Multilayer Perceptron (MLP), and Gradient Boosting

Machines (GBM). These algorithms have been widely used in various applications, including classification, regression, and clustering problems.

In this thesis, the machine learning algorithms are applied to a regression problem, specifically predicting high traffic conditions. These algorithms have been chosen for their ability to model complex relationships between input features and the target variable in regression tasks. By training and optimizing these models, our goal is to develop an accurate and reliable traffic prediction system that can effectively handle the challenges posed by the fog computing environment.

a. Random Forest: Random Forest is an ensemble learning method that constructs multiple decision trees during training and combines their output to improve the overall prediction accuracy and reduce overfitting. Random Forest handles missing values and outliers well, can model complex interactions between features, and is relatively robust to overfitting. Moreover, it provides a measure of feature importance, which can clarify the underlying relationships in the data. [6]

b. Multilayer Perceptron (MLP): MLPs are a class of feedforward artificial neural network that consists of at least three layers of nodes. The input layer, a hidden layer, A second hidden layer which is not essential and an output layer. Aside from the input layer, each node is a neuron that uses a nonlinear activation function. MLP utilizes a supervised learning technique called backpropagation for training the network. MLPs are flexible and can be used for both regression and classification problems. They can model complex non-linear relationships and are particularly effective when dealing with high dimensional data [7]. In This case, The MLP algorithm used has four layers of nodes, and is used to classify high and low traffic.

c. Gradient Boosting Machines (GBM): Gradient Boosting Machines are an ensemble learning technique that combines weak learners, typically decision trees, to form a strong learner. GBMs iteratively train weak learners, focusing on the errors made by previous learners in the sequence, and then combine their predictions using a weighted majority vote. This approach allows GBMs to model complex relationships between features and handle a wide range of data types. GBMs have shown excellent

performance in many machine learning tasks, but may require careful tuning of hyperparameters and can be prone to overfitting if not properly regularized. [8]

3.6 Edge, Fog, and Cloud Computing

Edge computing, fog computing, and cloud computing are different layers in a hierarchical computing architecture. Each of these layers serves different purposes, but they all work together to process and manage data in a distributed system.

The closest layer to the data source, is the Edge layer. This can be IoT devices, sensors, or other data-generating devices. In this layer, data is typically processed directly on the devices themselves or on nearby edge servers. This allows for almost real-time data processing and decision-making, reducing latency and bandwidth usage.

The next layer of the hierarchy is the fog layer. This layer acts as an intermediate layer between the edge and the cloud. It consists of fog nodes or gateways that aggregate data from multiple edge devices and perform additional processing and analytics. This layer can handle more complex computations than the edge layer and can make decisions about whether data should be sent to the cloud for further processing or discarded. Fogify, [3.6] operates primarily in this layer.

The uppermost layer is the Cloud Layer. This layer is the most powerful layer in the hierarchy. It is used for heavy-duty processing tasks, long-term data storage, and complex analytics that can't be handled by the edge or fog layers. Data from the fog layer is sent to the cloud for these tasks.

3.6 Smart Cities

Smart cities are taking advantage of digital technologies, data analytics, and connectivity to improve the quality of life for their residents, optimize resource utilization, and enhance the efficiency services. Examples of smart city applications include intelligent traffic management, waste management, energy conservation, and public safety.

As of this thesis, smart cities provide the motivation for developing traffic prediction and management solutions that can help to avoid congestion and reduce pollution. By taking advantage of edge computing and machine learning techniques, we aim to create a scalable, real-time, and accurate traffic prediction system that can be integrated into smart city infrastructures.

Edge computing has a crucial role in the context of smart cities, by enabling efficient data processing from IoT devices and sensors scattered throughout the urban landscape. This form of distributed computing architecture offers several key benefits to smart cities.

Firstly, it provides the capacity for real-time traffic analysis and prediction. By processing traffic data from sensors and cameras at or near the source, edge computing enables instantaneous analysis and prediction of traffic conditions. The reduced latency allows for swift responses to changing traffic scenarios, which aids in the prevention of congestion and reduction in travel times.

4. Methodology - pipeline

In this section, we outline the methodology employed in this thesis to develop our application - the edge computing-based traffic prediction and management system for smart cities that we deployed in Fogify framework emulator. The methodology consists of the following steps: Dataset and Scenario Creation, Implementation and Local Execution, Emulation and Scalable Deployment, Experimentation, and Evaluation.

4.1 Dataset and Scenario Creation

The first step in the methodology involves selecting an appropriate dataset, preprocess it to be suitable to use for machine learning, and creating a realistic traffic scenario.

To begin with, in data selection, a suitable traffic dataset is selected based on the available data columns. In order to develop accurate machine learning models, the dataset should have sufficient data points and cover a wide range of traffic conditions. The Metro Interstate Traffic Volume dataset [9] was chosen because it provided valuable information on

various traffic-related variables, such as hourly traffic volume, weather conditions, and time of the day, which were essential for the project.

Moving on, the selected dataset is cleaned and preprocessed to remove any inconsistencies, missing values, and outliers. This step ensures that the data is suitable for training and testing the machine learning models.

Lastly, a realistic traffic scenario is created based on the selected and preprocessed dataset. The scenario should simulate traffic conditions in a smart city, including factors such as peak hours, and weather conditions. By calculating the 75th percentile threshold for traffic volume, the model aims to predict whether the traffic volume will be above this threshold, indicating high traffic. This threshold was selected as it represents a significant deviation from the median traffic volume, therefore indicating a high level of congestion. This focus is representative of the traffic challenges faced by smart cities and allows for the evaluation of the effectiveness of an edge computing-based traffic prediction and management system in addressing these challenges. Through this, the model provides insights into potential traffic bottlenecks, demonstrating the value and potential applications of predictive modeling in a smart city context

4.2 Implementation and Local Execution

The next step in the methodology is the implementation and local execution of the project. This is performed using JupyterLab, the interactive development environment that was previously introduced. In this case, a .ipynb notebook is utilized to write and execute the code. The implementation is carried out using PySpark, a Python library for Apache Spark, which provides the ability to handle large datasets with ease.

The selected dataset undergoes several preprocessing steps to make it suitable for machine learning. This includes converting date and time to a timestamp, extracting hours from the timestamp, calculating the average traffic volume per hour, and creating a binary target variable for high traffic. The categorical columns in the dataset are processed using StringIndexer, which converts categorical features to numerical ones, a necessary step for machine learning algorithms. The continuous variables, on the other hand, are standardized

using the StandardScaler to ensure that differing magnitudes among variables do not affect the performance of the machine learning models. Following preprocessing, the dataset is split into training and testing sets, a standard practice in machine learning to evaluate the performance of the models.

The implementation makes use of a variety of machine learning algorithms, mentioned earlier. These algorithms were chosen for their diverse strengths and the ability to handle complex, non-linear relationships in the data. Each of these algorithms is trained on the training set, and their performance is evaluated on the testing set.

Performance evaluation is carried out by calculating the area under the Receiver Operating Characteristic (ROC) curve for each model. The ROC curve is a common metric used in machine learning to evaluate the trade-off between correctly predicting positive instances and incorrectly predicting negative instances. A higher area under the ROC curve indicates better model performance, allowing us to determine which algorithm performs best for our traffic prediction scenario.

4.3 Emulation and Scalable Deployment

In this phase, we utilize the Fogify and SparkEdgeEmu tools to emulate our edge computing-based traffic prediction and management system for smart cities and to deploy it at scale.

Fogify provides a testing bed for our edge computing solution, creating a simulated environment that can mimic various edge computing topologies. This environment enables us to assess the performance and efficiency of our solution in edge computing scenarios. The emulation process facilitated by Fogify captures crucial Apache Spark performance metrics, offering a holistic view of the application's behavior under different edge computing conditions.

Meanwhile, SparkEdgeEmu plays a pivotal role in the deployment of the Spark application within these emulated edge environments. It aids in the examination of different machine learning algorithms' performance within diverse edge computing topologies.

Once deployed within the Fogify Emulation Framework, our traffic volume prediction application undergoes a significant transition. From running on a single machine or a traditional cluster, it now operates within an emulated edge computing environment. This environment closely resembles a real-world edge computing network, complete with various edge devices, network links, and associated latencies and bandwidth constraints.

Throughout the emulation process, Fogify oversees the distribution of resources to the application's services, reflecting the edge computing scenario. These scenarios might include changes in network conditions, fluctuations in computational resources, or hardware failures. Fogify orchestrates the emulation environment and supports the gathering of metrics related to the Apache Spark application's performance and resource usage under these scenarios.

The emulation process provides critical insights into the application's behavior and performance in a realistic edge computing environment. This allows us to thoroughly test and optimize the application before deploying it in a real-world setting, thus ensuring it is prepared for the challenges posed by edge computing.

4.4 Experimentation

The experimentation phase is a critical part of the methodology, offering an opportunity to evaluate the performance of the traffic volume prediction application under various edge computing conditions. This stage is instrumental in assessing the application's performance when using different machine learning algorithms within a specific edge computing topology.

In the context of this project, the application is deployed within a particular edge computing topology using Fogify and SparkEdgeEmu. This approach facilitates understanding the influence of the network structure on the application's performance. The topology is designed to emulate real-world conditions in a smart city, exposing the application to an environment that includes varying network traffic, fluctuating computational resources, and dynamic load balancing across nodes.

The experimentation phase places significant emphasis on the collection and analysis of performance metrics. These metrics provide invaluable insights into the application's behavior under different conditions. Specifically, these metrics include used memory per

node, number of tasks per node, shuffle read per node, shuffle write per node, and duration in seconds per node. By examining these metrics, a comprehensive understanding of the application's performance within the emulated edge computing environment can be achieved.

Furthermore, the experimentation phase encompasses the evaluation of the application's scalability. This evaluation investigates how the application's performance adapts to changes in data volume or velocity. By analyzing the application's behavior under these conditions, crucial information regarding the application's capacity to handle increased loads is obtained.

Lastly, the experimentation phase includes an analysis of resource utilization. This incorporates a detailed assessment of how the application leverages key computational resources - specifically, CPU, memory, and network bandwidth - under the influence of varying machine learning algorithms. Through this process, it becomes possible to pinpoint areas of potential constraint and unearth avenues for subsequent performance enhancements.

Through careful experimentation, the robustness, performance, and suitability of the edge computing-based traffic volume prediction and management system for deployment in real-world smart city environments can be thoroughly assessed.

4.5 Evaluation

The evaluation stage of this project is designed to comprehensively assess the performance and effectiveness of the implemented traffic volume prediction application. It focuses on how well the application can predict traffic volume in a smart-city scenario, under the influence of different machine learning algorithms and within an emulated edge computing environment.

Firstly, the application's performance is evaluated using the Receiver Operating Characteristic (ROC) curve. The ROC curve represents the trade-off between the true positive rate and false positive rate, providing a quantitative measure of the model's classification ability. The area under the ROC curve (AUC-ROC) is then calculated for each machine learning algorithm used, offering a comparison of their respective performances.

Next, the evaluation process looks at how well the application operates within an emulated edge computing environment. Using the Fogify and SparkEdgeEmu tools, the application is tested under various edge computing conditions.

Performance metrics, including used memory per node, number of tasks per node, shuffle read per node, shuffle write per node, and duration in seconds per node, are collected and analyzed. This offers a detailed picture of the application's behavior under different conditions and reveals how efficiently it utilizes resources in the edge computing environment.

Finally, the evaluation process includes an analysis of resource utilization under the influence of different machine learning algorithms. This part of the evaluation helps identify areas where resources are over or underutilized, highlighting potential bottlenecks and areas for further optimization.

Overall, the evaluation stage provides a comprehensive assessment of the traffic volume prediction application's performance, and efficiency within an emulated edge computing environment. It offers valuable insights that can guide future improvements and optimizations, contributing to the development of a robust and effective traffic management solution for smart cities.

5. Implementation and Local Execution details

5.1 Import required libraries and initialize the Spark session

```
#Import all the necessary libraries
…
# Initialize Spark session
spark = SparkSession \
    .builder \
    .appName("pyspark-notebook") \
    .master("spark://spark-master:7077") \
    .config("spark.executor.memory", "512m") \
    .config("spark.sql.debug.maxToStringFields", 100) \
    .config("spark.driver.maxResultSize", "4g") \
    .config("spark.driver.memory", "8g") \
    .getOrCreate()
```

The first part of the code imports the necessary libraries for data processing, feature engineering, and machine learning tasks. It also initializes the Spark session. The Spark session is created with specific configurations like executor memory, driver memory, and others. The session is configured to handle the expected workload and resource allocation for the analysis. The Spark session is initialized with several specific configurations to optimize the application's performance. These configurations include:

spark.executor.memory: This setting defines the amount of memory each Spark executor process can use. The memory is set to 512 megabytes. This is used for the smooth execution of the application, to avoid out-of-memory errors.

spark.sql.debug.maxToStringFields: This setting controls the maximum number of fields to include in a string representation of a data frame or dataset. Here, it is set to 100. This configuration was useful because of the wide data frames that existed.

spark.driver.maxResultSize: This setting controls the maximum size of serialized results of actions. It determines the largest amount of data that the driver program can receive from the Spark executors. It is set to 4 gigabytes.

spark.driver.memory: This setting determines the amount of memory that can be allocated to the driver process. This is important for operations that require collecting data from executors to the driver. It is set to 8 gigabytes.

This initialization stage is vital as it determines how resources are allocated for the execution of Spark jobs and thus directly impacts the application's performance and efficiency.

5.2 Read and preprocess the data

```
# Read CSV file

path = os.path.abspath("data/Metro_Interstate_Traffic_Volume.csv")

data = spark.read.csv(path=path, sep=",", header=True)


# Convert the date_time column to a Timestamp type

data = data.withColumn("Timestamp", to_timestamp("date_time", "yyyy-MM-dd
HH:mm:ss"))


# Extract the hour from the Timestamp

data = data.withColumn("Hour", hour("Timestamp"))


# Calculate the average traffic volume per hour

grouped_data =
data.groupBy("Hour").agg(avg("traffic_volume").alias("Average_Traffic_Volume"))


# Calculate the 75th percentile threshold for traffic volume

threshold = grouped_data.approxQuantile("Average_Traffic_Volume", [0.75], 0)[0]


# Create a binary target variable for high traffic

data = data.withColumn("High_Traffic", when(col("traffic_volume") > threshold,
1).otherwise(0))


# Convert numeric columns to FloatType

numeric_columns = ["temp", "rain_1h", "snow_1h", "clouds_all"]

for column in numeric_columns:

    data = data.withColumn(column, col(column).cast(FloatType()))


# Use StringIndexer for categorical columns

categorical_columns = ["weather_main", "weather_description"]

indexers = [StringIndexer(inputCol=column, outputCol=column + "_index",
handleInvalid="keep") for column in categorical_columns]
```

The data from the 'Metro Interstate Traffic Volume' CSV file is read and initially processed. The 'date_time' column is converted into a timestamp format, and the hour component is extracted for further analysis. This aids in identifying traffic patterns during specific hours of the day.

Following this, an average hourly traffic volume is computed by grouping the data by the 'Hour' column. This approach enables a detailed understanding of traffic volumes across different time frames. A crucial aspect of this stage is determining the 75th percentile of the average traffic volume. This value acts as a threshold to distinguish between high and low traffic periods, thus facilitating the creation of a binary target variable, 'High_Traffic'. This binary variable, indicating whether traffic is high (1) or not (0), simplifies the classification problem, allowing machine learning algorithms to focus on predicting instances of high traffic.

For compatibility with the later stages of machine learning model development, numeric columns in the dataset are converted to the 'FloatType' data type. Machine learning algorithms commonly require numeric input in this format to function effectively.

Additionally, categorical columns in the dataset undergo transformation through a process known as 'indexing'. Utilizing the 'StringIndexer' function, categorical variables are encoded into numeric indices. This transformation is a necessary step in data preprocessing as machine learning algorithms require numerical input. By transforming these categorical variables into numerical indices, the machine learning models can leverage this information effectively in the prediction stage.

5.3 Prepare the data for machine learning

```
# Prepare the data for machine learning

input_columns = ["Hour"] + numeric_columns + [column + "_index"
for column in categorical_columns]

assembler = VectorAssembler(inputCols=input_columns,
outputCol="features")


# Create a pipeline with all preprocessing steps

pipeline = Pipeline(stages=indexers + [assembler])


# Split the data into training and testing sets

train_data, test_data = data.randomSplit([0.8, 0.2], seed=42)


# Fit and transform the data using the pipeline

pipeline_model = pipeline.fit(train_data)

train_data = pipeline_model.transform(train_data)

test_data = pipeline_model.transform(test_data)


# Create features_df

train_data = train_data.select(input_columns + ["High_Traffic",
"features"])

test_data = test_data.select(input_columns + ["High_Traffic",
"features"])
```

The next stage is to prepare the data for machine learning by assembling the input columns into a single features vector and creating a pipeline with all the preprocessing steps. This step simplifies the application of the machine learning algorithms as we can input them without modifying the data for each one.

Then, a pipeline is created to automate the preprocessing steps, which include transformations via StringIndexer and VectorAssembler. It ensures that the same sequence of transformations is consistently applied to both the training and testing datasets. This approach, reduces the errors due to the inconsistent data processing.

Then, data is splitted into training and testing sets. The training set is used to train the models, while the testing set is used to assess their performance on new data.

The pipeline then is fitted to the training data, and both training and testing data are transformed using the fitted pipeline model.

Finally, the features_df is created. This contains only the necessary columns from the transformed training and testing data, and its purpose is to keep the data organized and make it easier to work with when they are fed into the machine learning models

5.4 Scale the features

```
# Initialize the StandardScaler
scaler = StandardScaler(inputCol="features",
outputCol="scaled_features", withStd=True, withMean=False)


# Fit the StandardScaler on the training data
scaler_model = scaler.fit(train_data)


# Transform both the training and testing data using the fitted
StandardScaler
train_data = scaler_model.transform(train_data)
test_data = scaler_model.transform(test_data)
```

In this phase, the StandardScaler is utilized, to normalize the input features by scaling them to unit variance. This ensures that the features have the same scale, enhancing the stability and convergance of the algorithms during training.

Then, there is the transformation both training and testing data, using the fittedStandardScaler to obtain scaled features. This ensures the consistency and accuracy of the model of the predictions

5.5 Train and evaluate the models

The preprocessed data is used to train and evaluate machine learning models. Each model's training and evaluation process is outlined.

1. Multilayer Perceptron Classifier (MLP)
    a. Define the layers for the MLP model by specifying the number of neurons in each layer. Input layer, first hidden layer, second hidden layer and output layer.
    b. Initialize the MLP classifier with the specified layers defined before, the scaled features as input and the High_Traffic column mentioned before as the target variable.
    c. Train the MLP model by fitting it on the training data.
    d. Test the MLP model by making predictions using the trained MLP model over the test data.
    e. Evaluate the performance of the MLP model by calculating the area under the ROC curve. (AUC)

2. Gradient-Boosted Trees (GBT)
    a. Initialize the GBT model with the scaled features as input and the High_Traffic column as target variable.
    b. Train the GBT model by fitting it on the training data.
    c. Test the GBT model by making predictions using the trained GBT model over the test data.
    d. Evaluate the performance of the GBT model by calculating the area under the ROC curve. (AUC)
3. Random Forest Classifier (RFC)
    a. Initialize the RFC model with the scaled features as input, the High_Traffic column as the target variable, and a seed for consistent results.
    b. Train the RFC model by fitting it on the training data

c. Test the RFC model by making predictions using the trained RFC model over the test data.
d. Evaluate the performance of the RFC model by calculating the area under the ROC curve. (AUC)

The AUC represents the probability that a randomly chosen positive instance will have a higher predicted probability of being positive than a randomly chosen negative instance/ A higher AUC indicates better model performance.

5.6 Perform hyperparameter tuning

Machine learning pipelines require hyperparameter tuning to determine the optimal set of hyperparameters for a given model. Hyperparameters are model parameters that are not learned during the training process but are set by the user. They influence the model's learning process, capacity, and generalization ability. Tuning these parameters can lead to improved performance of the model.

In this project, hyperparameter tuning is performed using the CrossValidator class of the Spark ML Library. For each model, the hyperparameter grid and base model should be adjusted by fitting the crossValidator on the training data and retrieve the best model with the optimal hyperparameters. Then the best model is evaluated on the test data.

Each of the machine learning models used in this project, has a unique set of hyperparameters that control their learning process.

1. Multilayer Perceptron Classifier. The hypermeter parameters of the MLP, are the Number of Hidden Layers, the number of neurons in each hidden layer, the Activation function which can be the sigmoid, tanh, and the ReLU, the learning rate which determines the step size at each iteration while moving towards a local minimum of a gradient function and the Batch size which is the number of training examples utilized in one iteration.
2. Gradient Boosted Trees. The hypermeter parameters of the GBT, are the number of Trees (trees to be modeled), the learning rate which is the same as MLP, the maximum depth of a tree, the Minimum Samples per leaf and the subsample.

3. Random Forest Classifier. The hypermeter parameters of the RFC, are the number of trees, the maximum features to consider when looking for the best split, the maximum depth, and the minimum samples per leaf and split.

6. Emulation and Scalable Deployment details

The emulation of the project was performed using SparkEdgeEmu. The emulation allowed for controlled experiments, crucial for assessing the performance of the machine learning models implemented.

The emulation was a smart-city scenario. The smart city was designed with 9 nodes, distributed across three distinct “neighborhoods”. Each neighborhood represented a specific region of the city, with its unique traffic conditions and data generation rates. The nine nodes, including Raspberry Pi 3 and 4 models, and a small virtual machine, were chosen to represent a realistic heterogeneous edge environment, reflecting the variety of devices and computational resources that might be found in an actual smart city environment.

The emulation was designed to test the performance of the chosen machine learning models under different conditions. The metrics collected during the emulation included memory usage, number of tasks, shuffle read, task duration, and shuffle write for each node. These metrics provided invaluable insights of the machine learning models in predicting high traffic conditions. In the following sections, the metrics derived from the emulation results will be discussed further to give a deeper understanding of the system’s performance in the realistic edge computing environment.

Used Memory per Node: This is the amount of memory used by the application on each node (memory related metric). Memory usage is a crucial factor in performance, as insufficient memory can lead to issues such as increased disk I/O, slower processing speed, or even task failure.

Tasks per Node: This is the number of tasks that each node is assigned (CPU related metric). In Spark, a task is the smallest unit of work. This provides insights into the load balancing of the Emulation. Ideally, tasks should be evenly distributed among the nodes, for optimal performance.

Shuffle Read per Node: This metric is about the volume of data that a node fetches from its peers during shuffle operations (network related metrics). Shuffling is an operation in Apache Spark that redistributes data so that all the data needed for a particular computation is located on the same node. Such operations, although that they are necessary, most of the time they demand resources that exercise significant load on network I/O and time.

Shuffle Write Per Node: This metric represents the volume of data that a node writes to its storage during shuffle operations, destined to be fetched by another node (network related metrics). Like shuffle reads, shuffle write operations can be resource-intensive, with a high cost in terms of I/O and time. As a result, reducing shuffle write operations is a key strategy in performance optimization.

Duration Seconds per Node: This metric is the cumulative duration taken for all tasks to complete on each node (CPU related metric).

6.1 Data Preprocessing – Emulation Metrics

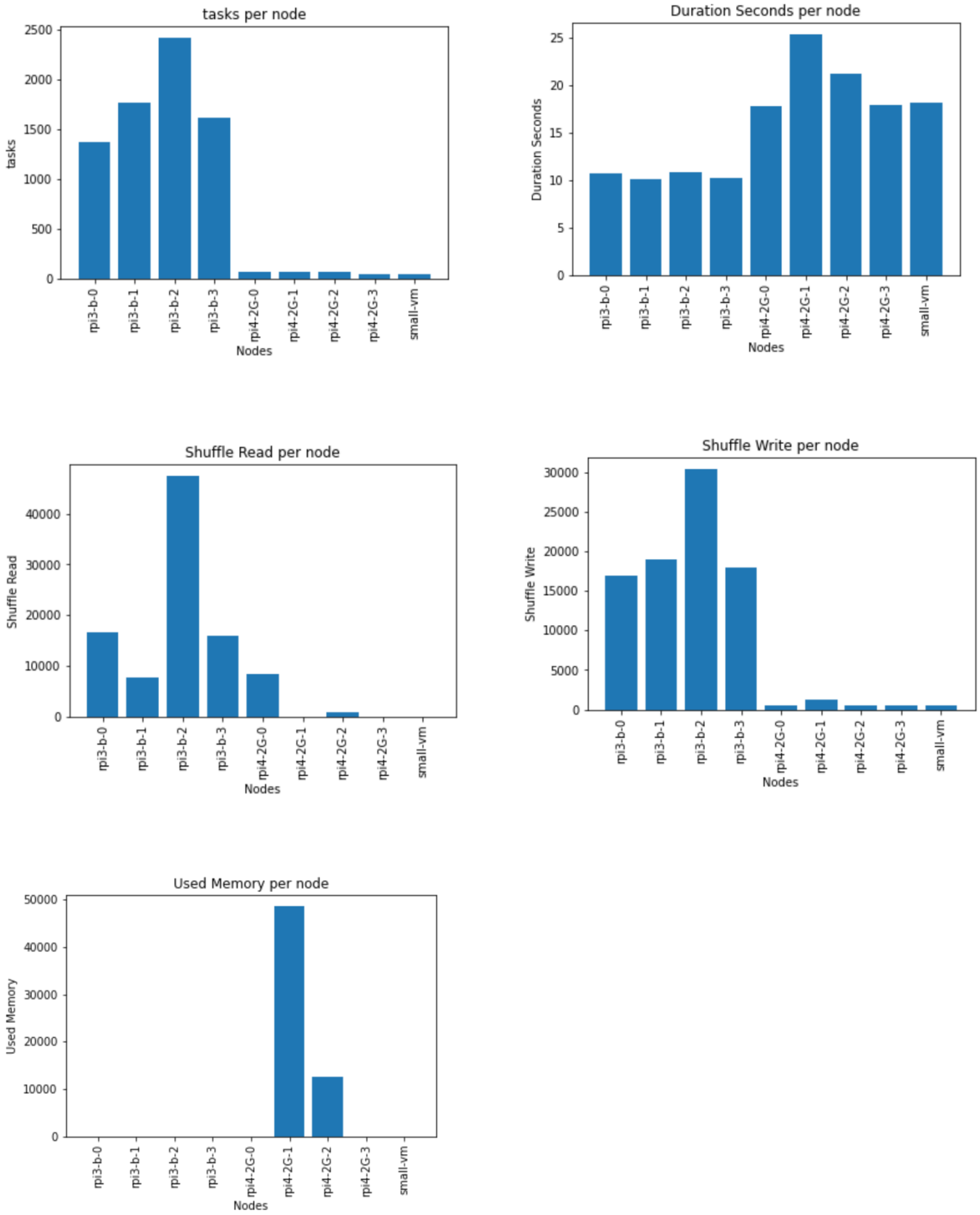


Figure 1: Emulation for preprocessing the data

Figure 1 shows the emulation results for preprocessing the data. The data preprocessing stage reveals interesting insights about the resource usage and task distribution across the nodes.

The Raspberry Pi 3 nodes (rpi3-b-0 to rpi3-b-3) handle significantly more tasks than the Raspberry Pi 4 nodes (rpi4-2G-0 to rpi4-2G-3) and the small VM. There appears to be a potential imbalance in task distribution, resulting maybe, in longer processing times. This maybe indicates the need for enhanced load balancing strategies.

Despite handling fewer tasks, the Raspberry Pi 4 nodes and the small VM require longer time durations to complete the tasks. This longer execution time could be attributed to factors such as hardware constraints, network latency, or the inherent complexity of the tasks assigned.

Shuffle read operations are prominent in the Raspberry Pi 3 nodes and one Raspberry Pi 4 node (rpi4-2G-0), signaling data movement between these nodes. However, the remaining Raspberry Pi 4 nodes and the small VM do not engage in shuffle read operations. This variation impacts the network load and overall processing time. Therefore, optimizing shuffle operations can potentially enhance performance.

The pattern repeats in the case of shuffle write operations, with Raspberry Pi 3 nodes, one Raspberry Pi 4 node (rpi4-2G-0), and the small VM involved. The data written by these nodes will be fetched by other nodes, affecting network I/O and potentially increasing processing times.

Regarding memory usage, only two nodes (rpi4-2G-1, rpi4-2G-2) report significant memory usage during the data preprocessing stage. The other nodes, including all Raspberry Pi 3 nodes, another Raspberry Pi 4 node, and the small VM, report no memory usage. This pattern might suggest that these two nodes are handling more memory-intensive tasks, or it could reflect specific memory allocation strategies.

In summary, these metrics offer a glimpse into potential areas for performance optimization, such as task distribution, task execution time, memory usage, and network efficiency.

6.2 Multilayer Perceptron Model – Emulation Metrics

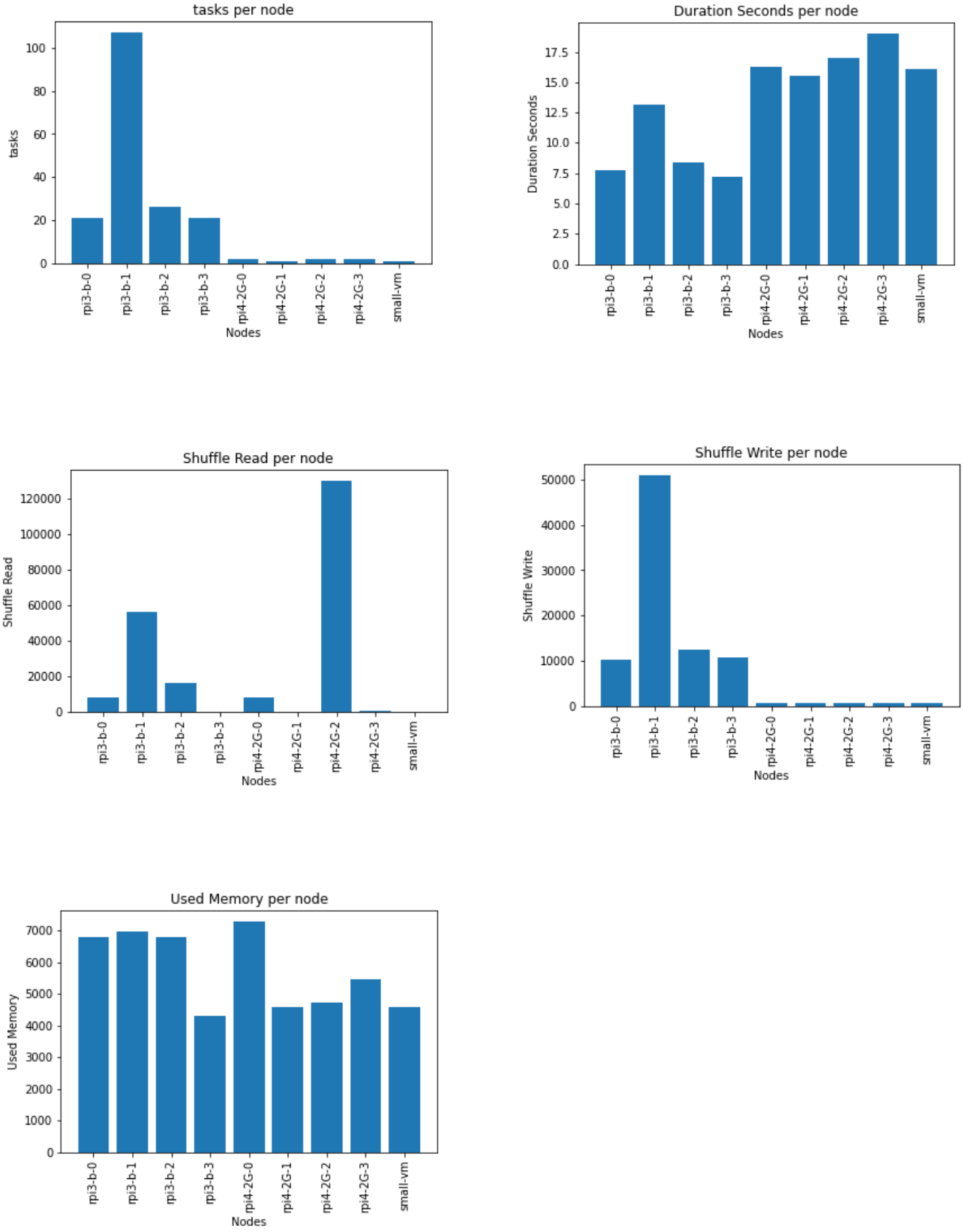


Figure 2: Emulation for Multilayer Perceptron Network

Figure 2 shows the emulation for Multilayer Perceptron Network. In the emulation of multilayer perceptron, the performance and resource usage reveal several points of interest. The execution of the model took 162 seconds, indicating a relatively high computational cost. Moreover, an error led to the loss of executor 0 on node rpi4-2G-2, likely due to the node exceeding resource thresholds or encountering network issues.

The task distribution among the nodes displays a stark disparity. Node rpi3-b-1 is burdened with the majority of the tasks, executing more than 100 tasks, while the other nodes handle significantly fewer tasks. This uneven distribution could potentially lead to bottlenecks and inefficiencies in the system.

The duration of task completion varies among the nodes as well, with the Raspberry Pi 4 nodes and the small VM taking longer than the Raspberry Pi 3 nodes. This longer duration on Raspberry Pi 4 nodes may reflect the inherent complexity of the tasks assigned, the hardware constraints, or network latency issues.

In the context of network operations, node rpi4-2G-2 shows a significant volume of data fetched from peers during shuffle read operations, while other nodes engage much less in these operations. This discrepancy could lead to an increased network load on rpi4-2G-2 and extend the overall processing time. Similarly, shuffle write operations are particularly high on some nodes (rpi3-b-0, rpi3-b-1, rpi3-b-2), indicating that these nodes are writing data to be fetched by other nodes, further affecting the network I/O and processing times.

Finally, memory usage across the nodes does not show drastic differences. However, the Raspberry Pi 4 nodes and the small VM use slightly more memory than the Raspberry Pi 3 nodes. This finding could be due to the nature of the tasks they are handling or to their specific memory allocation strategies.

In summary, this stage reveals several areas for potential optimization, including task distribution, task execution duration, network operations, and memory usage.

6.3 Gradient Boosted Trees Model – Emulation Metrics

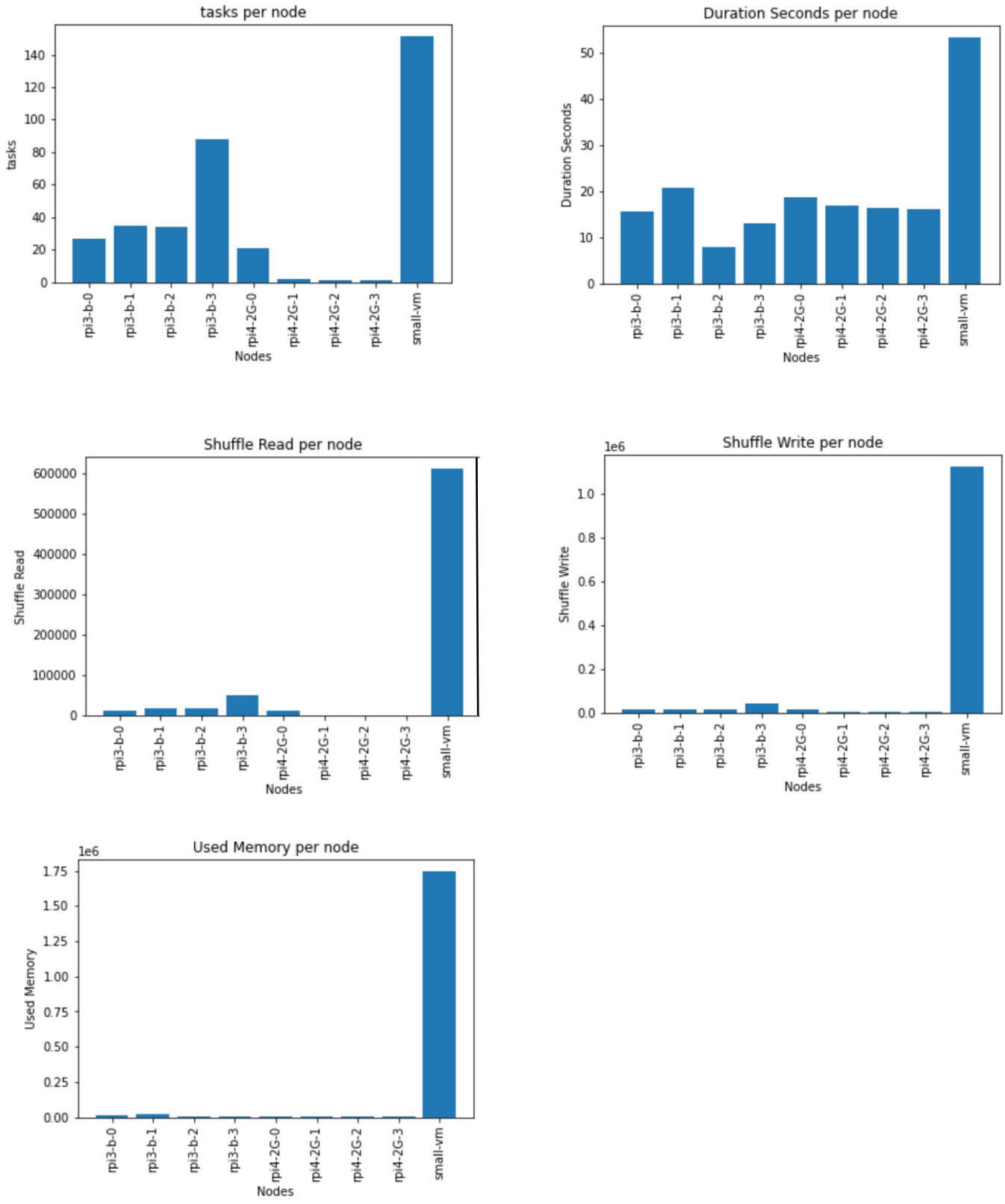


Figure 3: Emulation for Gradient Boosted Trees Network

Figure 3 shows the emulation for Gradient Boosted Trees Network. The gradient boosted trees model's emulation took 156 seconds to complete, and unlike the previous stages, no errors were reported.

The task distribution across the nodes in this stage was notably different from previous stages. The small-vm node was assigned a substantially higher number of tasks (about 150), while the other nodes were given considerably fewer tasks, with the lowest being two tasks for rpi4-2G-1 and rpi4-2G-2. This could lead to performance issues and could become a bottleneck in the system, considering the capacity of the small-vm.

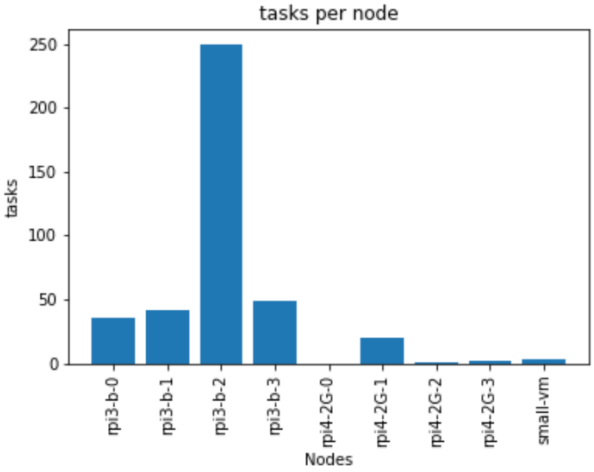
The completion time of tasks varied significantly across the nodes. The small-vm, assigned the highest number of tasks, took the longest duration (more than 50 seconds), which was considerably higher than the other nodes. This might be due to the large number of tasks assigned to it or other underlying system characteristics.

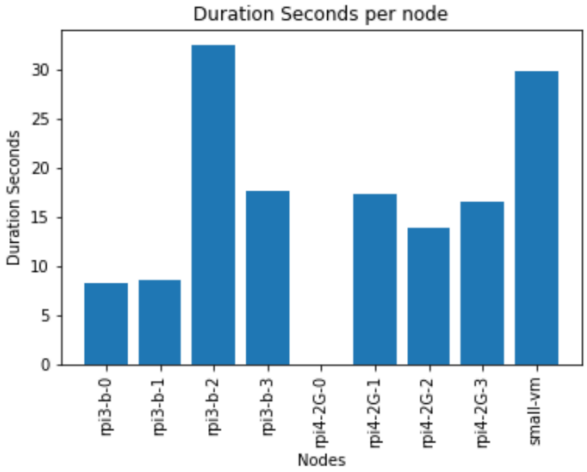
The shuffle read and write operations displayed a significant disparity across the nodes. The small-vm node experienced the highest volume in both shuffle read and write operations. This could place a considerable load on the small-vm and might be a factor contributing to its longer task completion time.

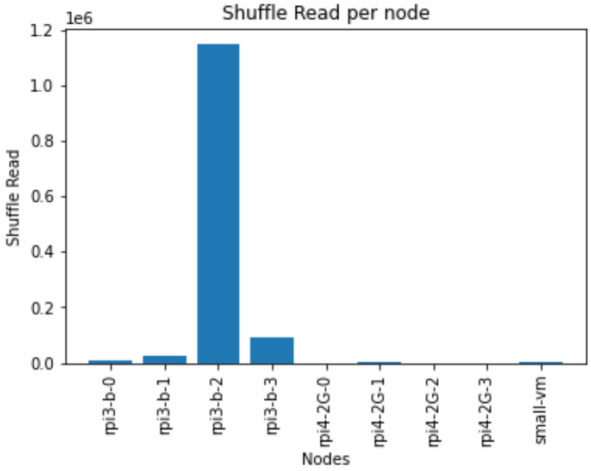
Memory usage was highest on the small-vm, reaching about 1,750,000 bytes, compared to the other nodes which had a memory usage between 15,000 and 25,000 bytes. Given the higher task load and increased shuffle read and write operations, the elevated memory usage on the small-vm is expected.

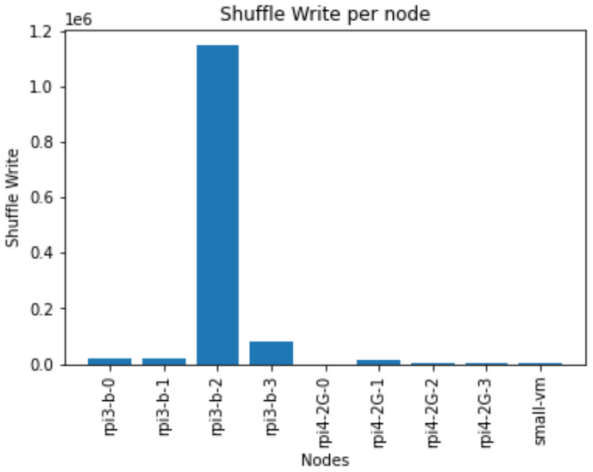
In summary, the gradient boosted trees stage displays an uneven distribution of tasks, considerable differences in task completion time, and significant disparities in shuffle operations and memory usage. This highlights potential areas for performance optimization and system enhancements, particularly regarding the small-vm, which seems to bear the brunt of the computational load.

6.4 Random Forest Model – Emulation Metrics









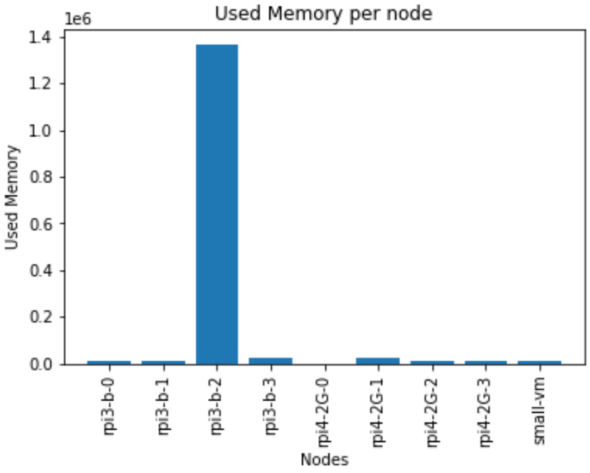


Figure 4: Emulation for Random Forest model

Figure 4 shows the emulation for Random Forest Model. The random forest model's emulation was completed in 137 seconds, however, an error was reported: the loss of executor 13 on rpi4-2G-0. This could be due to the node exceeding its capacity thresholds or experiencing network issues.

Task distribution was notably uneven across the nodes, with rpi3-b-2 assigned a disproportionately high number of tasks (about 250), while rpi4-2G-0 did not have any tasks assigned.

Task completion times varied widely across nodes, with the most noticeable delay on rpi3-b-2, taking about 32 seconds. This node was assigned the highest number of tasks, which likely contributed to the longer completion time.

Regarding shuffle operations, rpi3-b-2 demonstrated a significantly higher volume in both read and write operations, with about 1,150,000 and 1,200,000 bytes respectively, compared to the other nodes. This high volume of shuffle operations further indicates the high load on rpi3-b-2.

In terms of memory usage, rpi3-b-2 used a significantly higher amount of memory (about 1,400,000 bytes) compared to the other nodes. This, in combination with the high number of tasks and shuffle operations, indicates that this node is likely the most burdened in the random forest stage.

In conclusion, the random forest stage displays a marked imbalance in task distribution, shuffle operations, and memory usage, mainly concentrated on rpi3-b-2. The disassociation of rpi4-2G-0, which had no assigned tasks, raises questions about system stability and resource allocation efficiency. These factors present potential areas for system optimization and enhancements.

7. References

[1] "Docker Documentation," 2021. [Online]. Available: https://docs.docker.com/get-started/overview/.

[2] S. e. a. Turner, "Containers: A Complete Guide," *IBM*, 2017.

[3] M. e. a. Zaharia, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, 2012.

[4] X. e. a. Meng, "MLlib: Machine learning in Apache Spark," *The Journal of Machine Learning Research*, vol. 17, no. 1, pp. 1235-1241, 2016.

[5] S. e. a. Dhall, "Fog Computing: An Overview of Big IoT Data Analytics," *Wireless Personal Communications*, vol. 98, no. 1, pp. 353-380, 2018.

[6] L. Breiman, "Random Forests," *Machine Learning*, vol. 45, pp. 5-32, 2001.

[7] I. e. a. Goodfellow, Deep Learning, MIT Press, 2016.

[8] J. H. Friedman, "Greedy function approximation: a gradient boosting machine," *Annals of statistics*, vol. 29(5), pp. 1189-1232, 2001.

[9] J. Hogue, " "Metro Interstate Traffic Volume Data Set", UCI Machine Learning Repository," 2019. [Online]. Available: https://archive.ics.uci.edu/ml/datasets/Metro+Interstate+Traffic+Volume.