



**ASHESI UNIVERSITY**

**CONDITION MONITORING OF INDUCTION MOTORS USING IOT**

**CAPSTONE PROJECT**

B.Sc. Electrical and Electronics Engineering

**Edem Kwaku Bleboo**

**Agya Kwabena Ampomah Aboagye-Otchere**

**2024**

**ASHESI UNIVERSITY COLLEGE**

**CONDITION MONITORING OF INDUCTION MOTORS USING IOT**

**CAPSTONE PROJECT**

Capstone Project submitted to the Department of Engineering, Ashesi University College, in partial fulfilment of the requirements for the award of a Bachelor of Science degree in Electrical and Electronics Engineering.

**Edem Kwaku Bleboo**

**Agya Kwabena Ampomah Aboagye-Otchere**

**2024**

## DECLARATION

I hereby declare that this capstone is the result of my own original work and that no part of it has been presented for another degree in this university or elsewhere.

Candidate's Signature: .....

Candidate's Name: Agya Kwabena Ampomah Aboagye-Otchere

Candidate's Signature: .....

Candidate's Name: Edem Kwaku Bleboo

Date: .....

I hereby declare that preparation and presentation of this capstone were supervised in accordance with the guidelines on supervision of capstone laid down by Ashesi University College.

Supervisor's Signature:

.....

Supervisor's Name:

.....

Supervisor's Signature:

.....

Supervisor's Name:

.....

Date:

.....

## **Acknowledgments**

Glory be to God for helping us complete this report. We want to give special thanks to our supervisors, Dr. Stephen K. Armah and Dr. Robert Sowah, whose encouragement and academic advice helped us undertake this project. Additionally, we want to thank some people outside our supervisors who aided us in a meaningful way in our project. Acknowledgments to Bright Tetteh, Samuel Blankson, John Manful, Amma Oforiwaa, Kofi Sannie, Adim Anjouj, Desmond Kuuchi, Ayodeji Falase, Amanda Aninagyei, Kofi Agyapong and Emanuella Nketiah for playing a part in this project.

## **Abstract**

In today's rapidly evolving technological landscape, the convergence of the Internet of Things (IoT) and Machine Learning (ML) has paved the way for transformative innovations across various industries. One such critical application lies in electrical engineering, where detecting faults in electrical machines in real-time is paramount for ensuring operational efficiency and safety and minimizing downtime. In this report, a system is designed to monitor the condition of an induction motor.

This system measures voltage, current, temperature, and speed, which are then processed using Fast Fourier Transforms and ML techniques. The results were used to make predictions, facilitating real-time fault diagnosis and predictive maintenance of electrical machines. By making the results of the data analysis available on a website and through an on-site system health monitoring of induction machines is made more accessible and reduces the effects of reactive maintenance used in most industries.

This research paper includes the design processes, the implementation, results, discussion and conclusion. The results showcase our findings from the MATLAB Simulink model of an induction motor and the results of the data processing and Fast Fourier transforms. Additionally, the results of the machine learning module are highlighted and lastly the limitations and future works of this project was listed.

## Table of Contents

<b>DECLARATION.....</b>	<b>i</b>
<b>Acknowledgments .....</b>	<b>ii</b>
<b>Abstract.....</b>	<b>iii</b>
<b>Table of Contents .....</b>	<b>iv</b>
<b>Chapter 1.0: Introduction .....</b>	<b>1</b>
1.1 Background .....	1
1.2 Problem Statement .....	3
1.3 Objectives .....	3
1.3.1 Specific objectives.....	3
<b>Chapter 2.0: Literature Review .....</b>	<b>5</b>
2.1 Diagnosing Motor Malfunctions .....	5
2.1.0 Broken Rotor Bar Faults.....	6
2.1.1 Broken Rotor Bar Faults.....	6
2.1.2 Stator Faults.....	7
2.1.3 Rotor Eccentricity Faults .....	8
2.1.4 Bearing Faults.....	9
2.2 Condition Monitoring Techniques .....	9
2.3 IoT Involvement .....	11
2.4 Machine Learning Involvement .....	14
2.4 Design Choices and Criteria.....	18
2.5 Gaps Identified .....	19
<b>Chapter 3.0: Design.....</b>	<b>21</b>
3.1 Requirements Specifications .....	21
3.1.1 Functional Requirements.....	21
3.1.2 Non-Functional Requirements.....	22
3.2 System Design.....	23
3.2.1 Criteria for Selecting Modules Used for Hardware Design .....	23
3.2.1.1 Pugh charts for the microcontrollers .....	23
3.2.1.2 Pugh chart for the Communication technology.....	24

3.2.2 Block Diagrams .....	25
3.2.3 Hardware components .....	28
3.3 Circuit Schematic .....	32
3.4 Microcontroller and Battery Housing.....	33
3.5 Motor Selection .....	34
3.6 MATLAB and Simulink Simulation .....	34
<b>Chapter 4.0: Implementation.....</b>	<b>39</b>
4.1 Electrical System .....	39
4.1.1 Perforated Circuit Board.....	39
4.1.2 Sensor Placement .....	40
4.1.3 Arduino Code .....	43
4.1.4 Raspberry Pi Setup .....	43
4.2 Software System and Implementation.....	44
4.2.1 Database Design .....	44
4.2.2 API.....	47
4.3 Data Analysis Methods .....	49
4.3.1 Fast Fourier Transform.....	49
4.3.2 Machine Learning.....	49
4.3.2.1 Dataset used.....	49
4.3.2.2 Machine Learning Techniques .....	50
4.3.2.3 Imported Dependencies .....	51
4.4 Website Design.....	52
<b>Chapter 5.0: Results and Discussion .....</b>	<b>55</b>
5.1 MATLAB and Simulink.....	55
5.2 Electrical System .....	59
5.5 Data Processing .....	60
5.5.1 Fast Fourier Transforms .....	60
5.5.2 Machine Learning Module .....	62
<b>Chapter 6.0: Conclusion &amp; Future Work.....</b>	<b>64</b>
6.1 Conclusion.....	64
6.2 Limitations.....	64
6.3 Future Works .....	65
<b>References .....</b>	<b>66</b>

**Appendix .....68**



## **Chapter 1.0: Introduction**

This chapter introduces the problem of existing maintenance solutions for induction motors, and the proposed solution is condition monitoring of induction motors using Internet of Things (IoT). It establishes why the project is being performed and the importance of its objectives.

### **1.1 Background**

In the era of rapid technological advancement, the integration of the IoT has revolutionized various industries, and one such domain experiencing a paradigm shift is the condition monitoring of electrical machines. Electrical machines are pivotal in diverse applications, from industrial processes to household appliances. It is imperative to employ advanced monitoring techniques beyond traditional methods to ensure optimal performance, reliability, and longevity.

In electrical machines, condition monitoring involves continuously assessing parameters such as temperature, vibration, current, and voltage to detect early signs of deterioration or faults. IoT has opened unprecedented possibilities in this field, enabling real-time, remote, and comprehensive electrical machine monitoring. This integration facilitates the creation of an intelligent ecosystem where machines communicate, analyze data, and provide actionable insights, transforming how maintenance is approached [1]. This topic explores the synergy between condition monitoring and IoT, delving into the mechanisms, benefits, and challenges of real-time monitoring of electrical machines. By leveraging the power of IoT, industries can transition from reactive to proactive maintenance strategies, minimizing downtime, reducing operational costs, and enhancing overall system reliability [2]. Delving deeper into the intricacies of condition monitoring

using IoT for electrical machines, the project will uncover this convergence's transformative impact on the efficiency and sustainability of modern technological infrastructures.

Electrical machines are significant components in factories and various industries. They can be found anywhere, from schools to hospitals and more. There are two types of electrical machines this project focuses on monitoring. Generators and motors. Electric motors are widely used in industrial machinery, manufacturing, and automation. They drive conveyor belts, pumps, compressors, and other equipment, contributing to the efficiency and productivity of industries. Generators are machines that produce electrical energy and are used and needed by essential modern facilities. They supply power to hospitals, data centers, and more. Due to the involvement of electrical machines in most fields of human life, they must be reliable and work in optimum conditions. If these devices are faulty, they may affect transportation, power supply, automated systems, and other dependent systems. Failure of these machines could lead to catastrophic outcomes, so a real-time condition monitoring system must be utilized to ensure failure does not occur.

Electrical machines have several components that could fail, resulting in a reduced life cycle. Hence, it is necessary to implement a monitoring system to identify varying issues as early as they develop and determine what maintenance is required. Most industries wait for failure or very long intervals before servicing their machines. These long intervals are known as scheduled maintenance and can cost industries anywhere from 15% to 40% of production costs [2]. This is inefficient and costly because parts that could have been serviced earlier may be damaged entirely, and replacements would need to be bought. This costs industries more money because purchasing new parts is more costly than fixing damaged parts. By servicing at intervals, the technicians would be paid when their services may not be needed. Some typical faults in electrical machines are external short-circuit

faults, phase imbalance, overspeed, and pole slipping. Vibrations, smoke, and changes in current may characterize these faults. Instruments can measure all these, and by using IoT, such characteristics can be monitored.

## **1.2 Problem Statement**

Faults on electrical machines come at severe costs to many industries. Industries need electrical machines to operate efficiently and effectively. A faulty electrical machine could hinder factories' production, affect medical equipment such as ventilators and infusion pumps, and cause many more issues. An automated system capable of real-time monitoring and predictive maintenance of these electrical machines will help improve the life cycle of electrical machines and save industries time and costs. The automated system capable of this is an IoT system. IoT is a growing field with continuous improvements due to ongoing research by many experts working in the field. It provides a reliable method of collecting data and is quite flexible in its operations since many other devices can be added to the network seamlessly [3]. Using IoT and machine learning techniques to monitor the real-time parameters affecting an electrical machine's life cycle has countless advantages.

Today, induction motors are the most widespread in the industry, constituting more than 85% of all industrial motors [4]. Thus, there is the need for condition monitoring of induction motors with possible applications in other devices.

## **1.3 Objectives**

The main objectives of this project is to establish a fast and accurate fault detection system for induction motors.

### **1.3.1 Specific objectives**

1. **Real-time Fault Detection and Diagnosis:** The primary objective is to design an IoT system capable of noticing abnormalities and possible faults by real-time monitoring

of the various parameters, such as temperature readings, voltage and current levels, speed of rotation, and vibration readings that affect the electrical machines.

2. **Predictive Maintenance for Increased Reliability:** The project aims to develop predictive maintenance methods using data collected from the IoT system, incorporating trained Machine learning algorithms for predictive maintenance.
3. **Remote Monitoring and Management:** To build a user interface system for industrial users to monitor and manage remotely.
4. **Energy Efficiency Optimization:** Another key objective is to use IoT data to optimize the energy efficiency of electrical machines. The IoT system will be able to notice patterns of electrical machines' operations at different times, enabling users to identify areas where energy can be preserved.

## **Chapter 2.0: Literature Review**

This chapter aims to lay out an extensive overview of the related areas in the types of motor faults and research on predictive maintenance of electrical machines using a modern IoT system. This chapter will also explore related scientific papers, stating key findings and solutions and identifying gaps or areas for improvement.

### **2.1 Diagnosing Motor Malfunctions**

Faults in motors can occur at different stages of its life cycle. They can occur during the life cycle's design, manufacturing, or operation stage. The faults that occur in motors can be segregated into different categories: Electrical, Mechanical, Thermal, and External faults. These faults directly affect the performance of the motor via the stator, rotor, or bearings. Electrical faults are caused by insulation damage and partial discharge, overvoltage caused by the converter, significant fluctuations in power in inductive circuits, uneven voltage distribution in the windings, overvoltage in the insulation, common-mode voltages, and currents due to capacitive and inductive connection in the rotor, shaft, and bearings, high rate of voltage change  $dv/dt$  destroys the insulation [5]. Tangential and radial forces cause mechanical fault due to the magnetic field, non-uniformity of the air gap, vibration, and export from friction bearings [5]. Thermal faults are caused by mechanical overload, asymmetric power supply, a large number of consecutive releases, aging of the insulation, and poor cooling. Lastly, motor faults can also be caused by external factors such as high ambient temperature and excess dust, humidity, and acidity in the air [5]. Figure 1.0 showcases the percentage of faults caused by each main component that makes up an induction motor [6].

Table 1.0 shows that most faults are experienced in the stator and motor bearings. Seeing the various faults and causes of these faults and monitoring these machines can be

essential in optimizing their usage. Figure 1.0 also shows the breakdown of motor faults and their sources [7].

Table 1.0: Fault classification

Institute	Stator %	Rotor %	Bearing %	Others
IEEE	26	8	44	22
EPRI	36	9	41	14

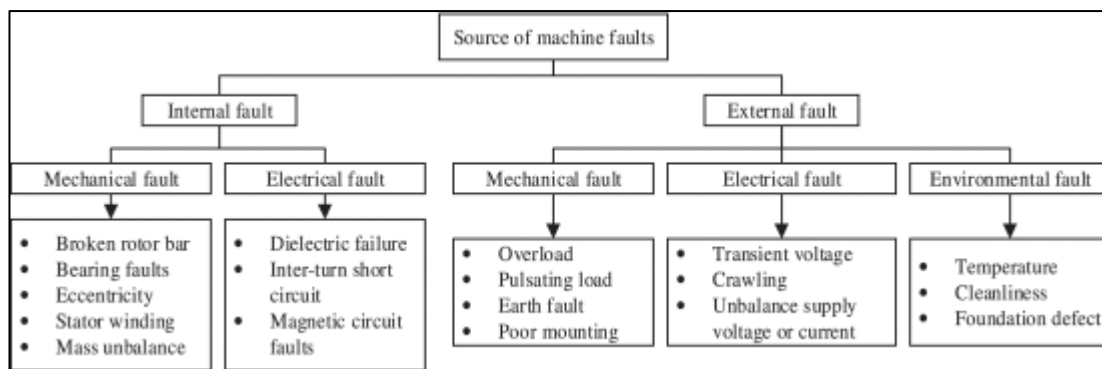


Figure 1.0: Machine faults sources

## 2.1.0 Broken Rotor Bar Faults

This section describes the different kinds of faults present in an induction motor—both electrical and mechanical faults.

### 2.1.1 Broken Rotor Bar Faults

A critical fault that can be found under mechanical faults is the Broken Rotor Bar (BRB). These are caused by simple manufacturing defects (air pocket, blow holes, improper alloy during manufacturing, weak point at the bar ending joint) or wear phenomena accelerated by internal constraints [7]. BRB fault can lead to unbalanced currents, torque positions, and decreased average torque [7]. The effects are usually more pronounced when several bars are broken. BRB fault can be identified when motor starting issues are due to

the inability to obtain sufficient starting torque [7]. Heat losses near the BRB increase significantly because the current passing through broken bars increases rapidly. Thus, by using thermal imaging, a BRB fault may be identified.

**2.1.2 Stator Faults**

As seen in Table 1.0, the stator accounts for the second most common fault detected in an induction motor. Stator windings consist primarily of electrical components, creating the rotating magnetic field needed for motor operation when an electrical current is passed through them. Most of the faults experienced in the stator are due to electrical faults, but some of these can occur due to mechanical faults in the stator. This division of mechanical and electrical faults in a star-connected stator can be seen in Figure 2.0. Mechanical faults from the stator frame or laminations can lead to electrical faults in the stator windings [7]. The various faults in a stator winding can be seen in Figure 3.0.

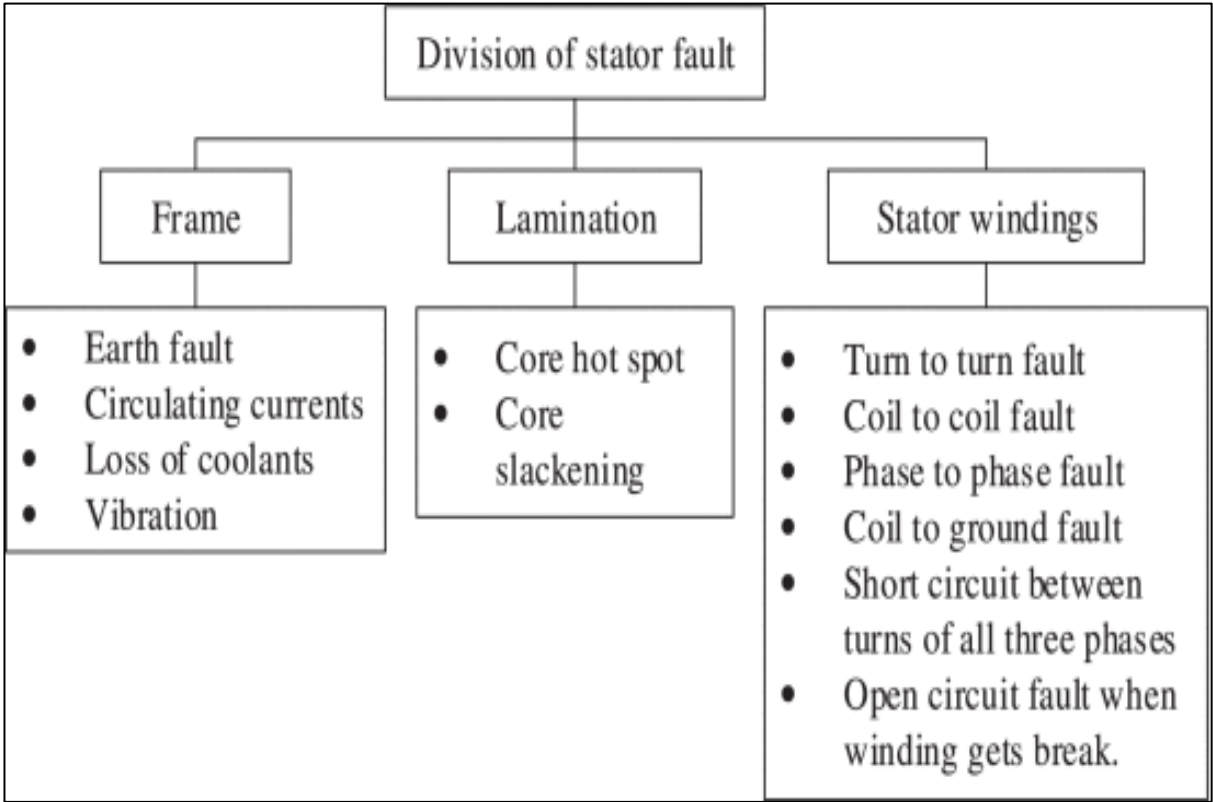


Figure 2.0: Stator fault sources

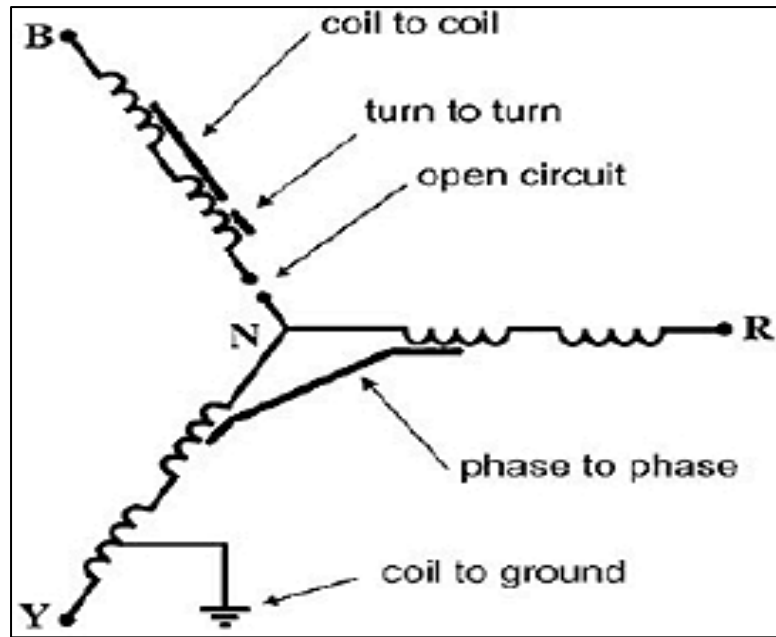


Figure 3.0: Stator winding fault sources

As seen in Figure 2.0 and Figure 3.0, stator winding faults in induction motors are essentially short-circuit faults. Short circuit faults can be identified by analyzing the unusual machine's operability. Short circuit faults induce unusual induction motor performance; the increase in vibration can cause a rise in torque pulsation and a decrease in the average torque [7]. From this paper, the stator winding short circuit fault could be diagnosed by collecting data on the vibration and torque.

### 2.1.3 Rotor Eccentricity Faults

These are faults caused by improper manufacturing and assembly of the motor, such as misplacement of bearing misalignment between the rotor shaft and load axis [7]. Rotor Eccentricity may lead to unbalanced magnetic pull and can be identified by asymmetrical operation, vibration, and electromagnetic noise. They may also cause bent rotor shafts, bearing wear and tear, and, eventually, breakdown. It can be identified by high-order harmonics when analyzing the current spectrum. The torque and speed of a motor fluctuate due to significant harmonics imposed on the stator current.



#### **2.1.4 Bearing Faults**

This common fault in Induction motors is caused by wear of the bearings. Prolonged wear of bearings leads to a complete mechanical breakdown of the motor. Bearing faults account for about two-thirds of induction motor faults [7] and must be considered in any health monitoring of induction motors. Bearing faults are almost impossible to reproduce together with others on a test bench experimentally. Hence, it is a challenge to obtain a data set with bearing defects and other faults coinciding [8]. In the paper, BD examples were artificially provoked by drilling a through-hole on the outer race with a 1.191 mm tungsten drill bit.

#### **2.2 Condition Monitoring Techniques**

Considering the variations of faults in mechanical machines, several techniques are available to evaluate their condition. Some standard methods are vibration monitoring, chemical analysis, temperature measurement, chemical analysis, temperature measurement, acoustic emission monitoring at ultrasonic frequencies, sound measurement, laser displacement measurement, and stator current monitoring, among others [9]. Bearing failures account for approximately half of all machine failures, so it is prudent to consider the state of bearings above most other components. Vibration monitoring is a common technique involving using accelerometers or vibration sensors attached to the bearings or structural components affected by the machine's vibration during operation. Vibration spectra analysis has been proven to be the most effective health monitoring and diagnostic method for rotating machinery and was used by [10] to determine the condition of the bearing. Techniques based on vibration signals give a fault diagnosis focused on the location of spectral components associated with [11].

Temperature monitoring, which is also common, is a vital technique because it can be used to monitor the state of both the windings and bearings. It can be done with

thermistors or an infrared camera that determines the hot regions of the device. In [8], the method of area selection of image differences (MoASoID) feature extraction using the thermal images method was used. This compares many training sets together and selects the areas with the most significant changes for the fault recognition process. A pseudo-coloring technique on an infrared image of the motor can be applied to the motor after it reaches a thermal steady state. A color-based segmentation technique was used to identify abnormal hot regions in the thermal image [11].

Temperature monitoring using embedded sensors like thermistors may be challenging to install, and further investigation must be performed to determine the cause of the temperature rise. Ultrasonic frequency monitoring, a costlier method than the rest, can be used instead of vibration monitoring. This is because the stress waves at high frequencies emitted by the bearings of electrical machines can accurately be read to evaluate the types of faults that have occurred [9]. This may be more efficient than vibration and lower-frequency sound monitoring because there will be less ambient interference, which means a higher signal-to-noise ratio.

Voltage monitoring, which is not used as much as other methods, involves reading the device's voltage through intrusive means to determine its health. Due to the intrusive nature of the technique, it is not as favored in designs that are meant to be portable. It also is not a great way of determining the state of a device. In [8], each phase's voltage was measured and used with the Principal Component Analysis statistical method and classification trees to identify faults.

Furthermore, rotational speed monitoring is another technique that can be used to determine the state of an induction motor. In [8], speed monitoring was performed using a digital encoder. In induction motors, the speed and voltage are directly proportional. Thus,

by changing the voltage, the speed is also changed. This relation can be used in the monitoring process to determine device health.

Finally, stator current monitoring is an easy method that can be implemented to monitor the condition of electrical machines. The current going to or from generators and motors can be measured using standard transformers and intermediate devices. Thus, the sensors must not be attached directly to the machines. Current sensing is not advisable because motor faults mostly exhibit a minuscule effect on the current. Mechanical and electrical failures produce vibration in electric motors with different amplitudes and frequencies. Thus, solutions monitoring the health of motors mainly focus on measuring vibration and temperature [12].

### **2.3 IoT Involvement**

In traditional industrial settings, wired sensor networks are connected to programmable logic controllers (PLCs), which connect to a primary data bus and a human interface [13]. A disadvantage of this arrangement is that too many wires are used, which wastes resources because wired networks will not use as much material. Also, the traditional methods do not perform data analytics to predict fault occurrence. Due to the inefficiency associated with conventional methods, IoT technology must be integrated into industrial settings. IoT brings many advantages and improvements, such as enabling data to be stored and processed in the cloud for predictive maintenance and monitoring device run Time and efficiency. Furthermore, it saves resources by cutting down on conductor material use, which would require more time and money. IoT systems are also versatile and much more scalable than the traditional method since other devices can be seamlessly added to the existing infrastructure without creating numerous wired connections that would clutter the setup.

In current settings, two systems are used to induce motor health monitoring. The first is the direct installation of sensors by manufacturers, which requires specific software for analysis, and the second is the use of portable devices with sensors that also use specific software and are considerably costly [12]. The idea is to create a low-cost system that can be used more generally and with an accessible, accurate analysis system.

Since IoT systems are low-power and lossy networks, serious considerations must be considered for the communication protocol to obtain an efficient system. In [11], the IoT protocol stack comprises 802.15.4, 6LoWPAN, RPL, and CoAP to monitor the temperature and vibration of several pumps. 6LoWPAN offers end-to-end IP addressable nodes without a need for a gateway. Still, it poses security issues and is less immune to interference than wifi and Bluetooth devices. It also has a short range, which may be unfavorable. CoAP is a protocol based on UDP with a request/response model. It is an efficient RESTful protocol with low latency and power consumption but is unreliable due to the use of UDP, has a long processing time, and is unencrypted.

In [9], a multi-sensor module was used at the edge layer alongside Raspberry Pi four as the gateway. The two devices communicated through Bluetooth Low Energy, and the Raspberry Pi used the MQTT protocol to share data with the cloud device. The Raspberry Pi 4 obtained 65536 samples of vibration data, which was enough revolutions to cover the dynamic behavior of the motor. This allowed the data to be processed using Fast Fourier Transforms (FFT) from the SciPy python library on the Raspberry device. The long transmission time using Bluetooth was an essential factor that influenced data collection and the limit in the number of multi-sensors that could connect the gateway. Figure 4.0 and Figure 5.0 show the setup of their system.



Figure 4.0: Network design

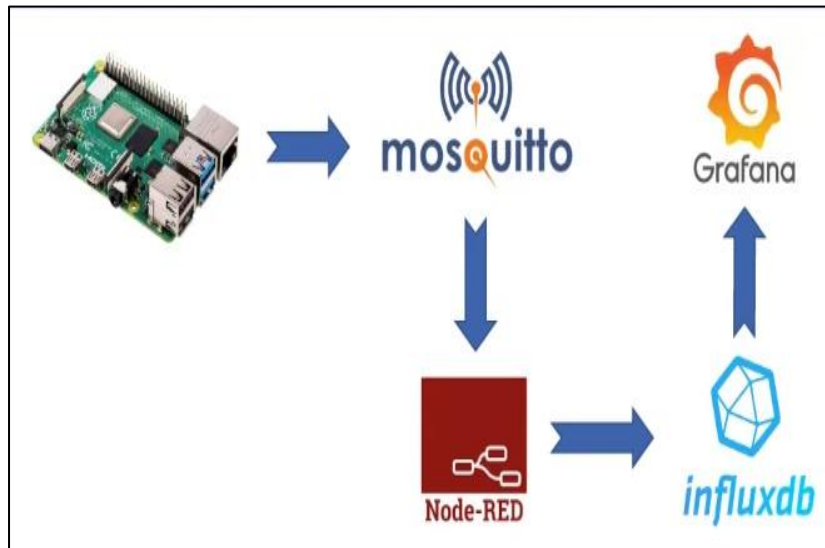


Figure 5.0: Cloud Design

In [12], a sensor node consisting of an accelerometer was connected to the control unit, an ESP32 – WROVER. The choice of a control unit provided the ability to temporarily store large amounts of data. Data collection was performed every sixty minutes with a four-minute delay before transmission. The transmission was done through Wifi using the MQTT protocol. MQTT offers a lightweight, low-bandwidth protocol used in IoT. It works using a publisher-subscriber model and provides greater scalability for the IoT network. It is efficient and reliable and thus may be an excellent choice for this project.

## 2.4 Machine Learning Involvement

Machine learning is a valuable technique that can be utilized in the condition monitoring of induction motors. This technique is relatively new but has proven effective, so artificial intelligence and machine learning are ever-growing in the condition monitoring of induction motors. Different artificial intelligence methods were utilized in fault detection and diagnosis, and they were all compared. The techniques used were Naive Bayes, k-nearest Neighbor (KNN), Support Vector Machine (Sequential Minimal Optimization) (SVM/SMO), Artificial Neural Network (Multilayer Perceptron) (ANN/MLP), Repeated Incremental Pruning to Produce Error Reduction, and C4.5 Decision Tree [16].

After these methods were compared, it was found that the ANN/MLP and KNN performed with 100% accuracy for stator faults. For broken rotor bar faults, the ANN/MLP and KNN methods had an accuracy of over 99.7%, and lastly, for bearing faults, the KNN, ANN/MLP, and C4.5 methods showcased the best performances. In all, for multi-classification of the various faults, the ANN/MLP and KNN methods proved to be the best options, with accuracies reaching over 92.5%. The results and comparisons showed that ANN had the best fault detection performance for various faults that an induction motor could experience [16].

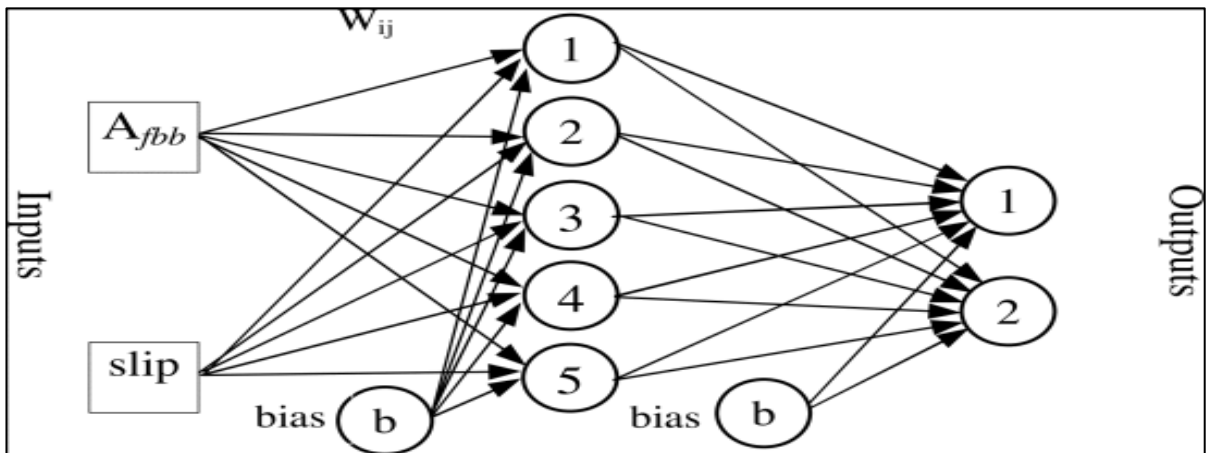


Figure 6.0: Neural network architecture

Figure 6.0 represents the Neural network architecture for the ANN technique's broken rotor bar fault detection. The network hosts two input neurons, a hidden layer of five neurons, and two output neurons, 1 for motor state and 2 for load range. Before building the neural network and applying the ANN technique, the appropriate input variables, the number of output variables, and the layers and neurons in each layer were all considered.

The two input variables selected were the harmonic amplitude associated with the broken rotor bar and the rotor bar slip, which was used to improve the fault classification. The output neurons selected were the motor state and load range [16]. The results from the output neurons utilized a sigmoid activation function, meaning that the results ranged from 0 to 1. With the load range output, neuron 0 referred to the case of low loads, 0.5 referred to high loads, and 1 referred to dull load conditions. On the other hand, the motor state output neuron and output of 0 referred to a healthy machine, 0.5 referred to one broken bar, and one referred to two broken bars. It was found that the results of this ANN technique for rotor bar fault detection were acceptable after a few tens of iterations, as seen by the Figure representing the training data results provided in Figure 7.0 [16].

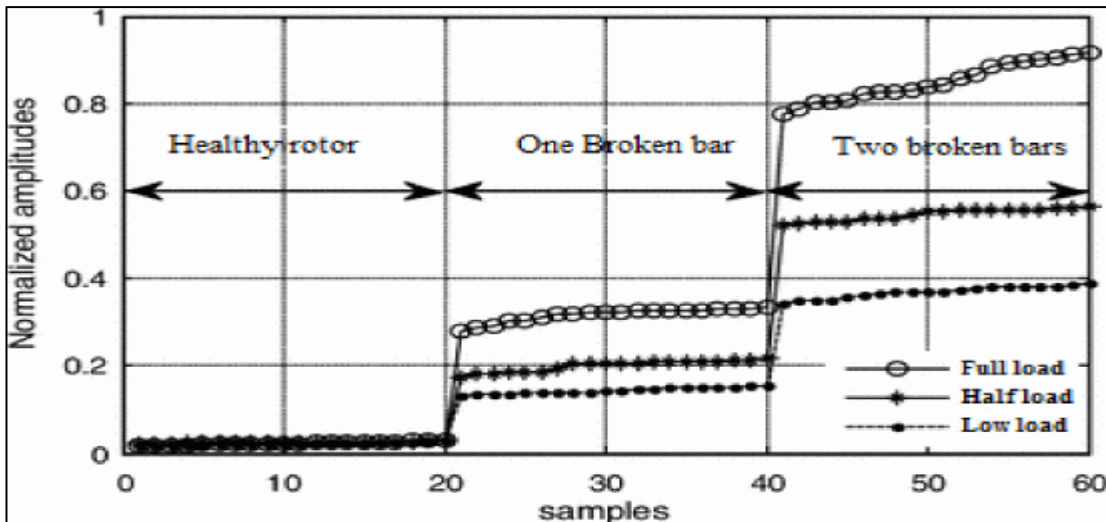


Figure 7.0: Normalized amplitudes of broken rotor bar components at different loads and motor states

In [17], the results of the ANN on the test set data. Four thousand epochs were run, and the results were beyond satisfactory. The results were 100 percent acceptable and represented as shown in Figures 8.0 and 9.0.

Figure 8.0 represents the output of neuron one, which is associated with the motor state. A value of 0 represents a motor in a healthy state, 0.5 represents one broken bar fault, and 1 represents two broken bar faults. From Figure 8.0, we can see that each motor state output correlates with the state of the motor [17]. All the samples of the healthy machine results were around the accurate value of 0, 1 broken bar result was around the accurate value of 0.5, and 2 broken bar results were around the accurate value of 1. The second output neuron, the load range, also gave accurate results that were also 100% acceptable. This can be seen in Figure 9.0, where the samples of the low-load machine results were around the accurate value of 0, half-load results were around the accurate value of 0.5, and full-load results were around the accurate value of 1 [17].

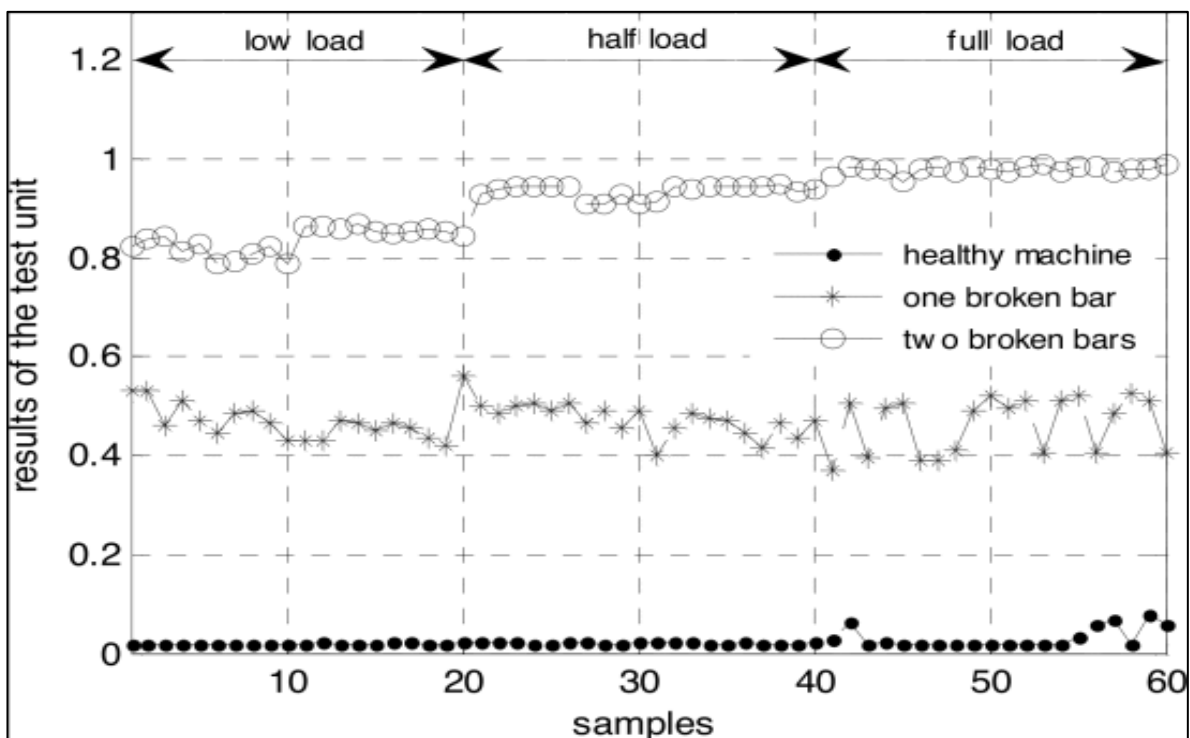


Figure 8.0: Motor state results of the test unit



Utilizing machine learning techniques, as seen, can be a nonintrusive method of conditional monitoring, with accuracy above 90%, depending on the method chosen. It highlights how it should be incorporated into any condition monitoring technique as a form of redundancy or on its own. From [17], machine learning techniques are growing in terms of their use cases in condition monitoring of machines. Still, it failed to highlight any existing issues or concerns in utilizing this technique.

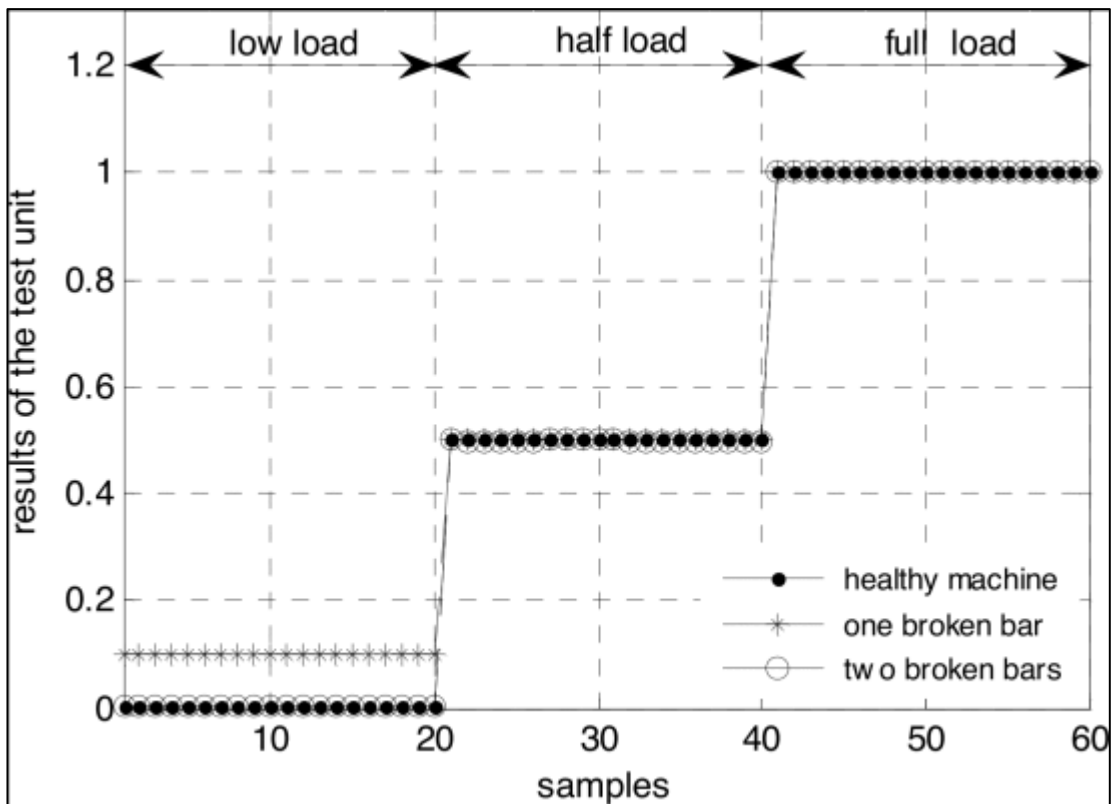


Figure 9.0: Load range results of the test unit

In [8], a multi-fault diagnosis system was used in the health monitoring of electrical machines using multiple sensors for single, combined, and simultaneous faulty conditions. The relevance of this research stems from the possible occurrence of numerous simultaneous faults in induction motors. In this study, a statistical technique, Principal Component Analysis (PCA), is combined with a Machine Learning technique, Decision Trees, to solve

a multiple prediction problem, commonly known as multi-label classification. PCA is used for exploratory analysis and is a popular method for dimensionality reduction [8]. A decision tree is an algorithm in which a set of tests, organized hierarchically, is used to guide the process of class assignment or output value calculation. The method uses comparison based on recorded attributes to direct branches until a leaf node with no further branches is reached, where a class is assigned to the instance. The process starts at a root node and is divided into subsets based on the different cases. An instance can belong to several classes simultaneously, or several attributes must be predicted simultaneously, known as multi-label classification. Using this method, the research successfully classified unbalanced and misaligned faults using vibration and current data [8]. This was initially a challenge because the unbalanced condition produced a larger vibration amplitude at  $(1 \times \text{RPM})$ . In comparison, the misalignment condition produced more radial and/or axial vibrations that usually produce their dominant frequency components at  $2 \times \text{RPM}$  and  $1 \times \text{RPM}$  [8]. From the study, it is noted that better results could be achieved by training several models to deal with each fault, even though classification trees had the best accuracy.

## **2.4 Design Choices and Criteria**

Conditioning monitoring of electrical machines has different stages, and each requires its own specifications. These stages consist of the sensors used to monitor the electrical machines and IoT implementation, including the microcontroller and the cloud service. The goal of most researchers was to use low-cost but accurate sensors, which were preferred. This resulted in a Negative temperature coefficient (NTC) thermistor (Amphenol, JS5698) chosen due to its high heat transfer, long-term stability, cost, and ability to be attached to the inside or outside of the machine case. This choice of temperature sensor was made based on options such as the LM35 temperature sensor, other thermistors, and

thermocouples [13]. A vibration sensor is one of the essential sensors used in fault detection. The sensor chosen from the "Condition Monitoring of Electrical Machines with the Internet of Things" research paper was a piezo cantilever-type vibration sensor (TE Connectivity, MinSense 100) due to its high linearity, low cost, dynamic range, and size. Other research papers made use of different sensors for the condition monitoring process. A tri-axial accelerometer (AXDL345) was used as a vibration sensor, and a multi-sensor module with eight integrated sensors was used. This multi-sensor module was expressed as a low-cost module with eight sensors, such as an accelerometer, a magnetometer, a humidity sensor, and more [1],[12].

## **2.5 Gaps Identified**

Each paper under study investigated the task of condition monitoring using a unique approach. There are a few areas where improvements can be made. For example, a friendly user interface can be developed for end users in charge of machine operation. This interface should account for numerous machines. Furthermore, most papers focused on obtaining the data and determining when faults occur. Fault prediction is an area that is still being investigated using various methods. It would also be prudent to consider edge and fog computing compared to cloud computing. This helps achieve the goal of creating a low-cost system that is more secure because there is no direct access to the Internet. Finally, most research did not consider identifying when multiple faults coincided and the prediction of the occurrence of numerous faults.

In conclusion, the literature review provides a thorough overview of the conventional and emerging technology associated with predictive maintenance for electrical machines using IoT systems, covering diagnosing motor malfunctions, condition monitoring techniques, IoT integration, machine learning involvement, and identified gaps for improvement. It expounds on the diverse kinds of motor faults, explores various

monitoring methods for timely fault detection, and highlights the advantages of IoT integration, including cloud-based data processing and resource savings. It displays the current gaps in proposed solutions, such as the unaddressed need for user-friendly interfaces and advancements in fault prediction, underscoring the ongoing efforts required to optimize maintenance practices effectively. It also expounds on machine learning tools that are applied in condition monitoring.

## **Chapter 3.0: Design**

This chapter explains the choices in designing the envisioned system and expounds on the system architecture. The explanations are assisted with tables describing the design choices and flow diagrams showing how the system operates. Furthermore, this chapter introduces the circuit schematic, the Simulink simulation that was conducted, and the prototype for the ESP32 housing.

### **3.1 Requirements Specifications**

The necessary functional and non-functional requirements to fulfill the proposed design objectives for the condition monitoring system are listed in this section.

#### **3.1.1 Functional Requirements**

This section outlines the essential functional requirements for the system's hardware and software components. These requirements ensure the system's autonomy, efficiency, and robustness in monitoring and managing electrical machines. The functional requirements for the hardware and software aspects of the system are as follows.

##### **Functional Requirements for Hardware Side**

1. The system should be fully autonomous.
2. To maximize battery life, the system must take parameter readings for four minutes between fifteen-minute intervals, during which it will sleep.
3. Data should be uploaded to the cloud database every minute.
4. The system should be capable of identifying faults in electrical machines based on the monitored parameters.
5. The system should be robust and run for months without direct human interference.

6. The system must use a suitable communication protocol for data transmission and processing.
7. The system should be energy efficient.

Functional Requirement for software side:

1. The system has an interface that shows results from data analysis, such as fault detection, prediction of faults, and prescriptive measures.
2. The interface needs to be structured to accommodate large networks.
3. The machine learning module should be able to make predictions efficiently.
4. The system should be secure.
5. The data analysis should be done at the edge level and can be uploaded to the company network.

### **3.1.2 Non-Functional Requirements**

This section delineates the non-functional requirements that the system must meet to ensure its usability, compatibility, and reliability. These requirements address the system's integration, performance, and adaptability to various conditions and environments. Below are the specified non-functional requirements:

1. The system should blend well with the existing infrastructure.
2. The dashboard should show updates, and a graph layout of the readings collected and be user-friendly.
3. The system must operate reliably under various environmental conditions.
4. It should be scalable to accommodate many electrical machines and sensors without compromising performance.

5. It should be compatible with different types and models of electrical machines and sensors commonly used in industrial settings.

### 3.2 System Design

#### 3.2.1 Criteria for Selecting Modules Used for Hardware Design

Criteria for selecting a Microcontroller: Two microcontroller units would be used to model a network with multiple devices that would need several devices to be monitored. One kind would be connected to each machine, and the other would be the main gateway. The first controller must be cheap, small, fast, and energy efficient. It should also have suitable wireless communication capabilities to be reliable and energy efficient.

##### 3.2.1.1 Pugh charts for the microcontrollers

This Pugh chart analysis in Table 2.0 concluded that the ESP32 microprocessor would be the most suitable based on the specified criteria for the first microcontroller. Its cost-effectiveness, robust nature, and high processing speeds made it the best option relative to its competitors.

Table 2.0: Pugh chart for the first microcontroller

Criteria	Weight	ESP32/Baseline	Arduino nano BLE	Raspberry Pi	ESP8266
Cost-effective	4	0	-1	-3	1
Convenience (Size)	4	0	0	-3	0
Wireless communication	3	0	-1	1	0
Processor speed	5	0	-1	3	-1
Bluetooth	5	0	0	1	-1
Robustness	3	0	-1	-1	0
<b>Total</b>	<b>24</b>	<b>0</b>	<b>-4</b>	<b>-2</b>	<b>-1</b>

Table 3.0: Pugh chart for the second microcontroller

Criteria	Weight	Raspberry Pi (Baseline)	Arduino Uno	ESP32	STM32
Cost-effective	2	0	2	2	0
Processing power	5	0	-3	-3	0
Wireless communication (Wifi)	4	0	-3	1	-3
Processor speed	4	0	-3	0	2
Bluetooth	2	0	-3	1	-1
Robustness	3	0	2	-2	1
<b>Total</b>	<b>20</b>	<b>0</b>	<b>-10</b>	<b>-1</b>	<b>-1</b>

The second microcontroller needs to be powerful at processing data. This is because the machine-learning model at this edge layer will analyze and evaluate the data from several electrical machines. The microcontroller should also have ample storage space for the same purpose. It should have wireless capabilities to communicate with the company network and remote devices. It does not need to be very energy efficient, so the communication method must be flexible. The microcontroller should be almost always active to receive data from other devices in the network. The Pugh chart analysis in Table 3.0 showed that the Raspberry Pi is the best choice.

### 3.2.1.2 Pugh chart for the Communication technology

In the Pugh chart analysis in Table 4.0, Wi-Fi was proven to be the communication technology for this project. Relative to LORA and Bluetooth, its low latency and data rate specifications proved more than sufficient for the project. Utilizing Wi-Fi as communication technology enables using the MQTT protocol, which is optimized for low bandwidth, high latency, or unreliable networks. It is also lightweight with minimal overhead and is ideal for frequent small data transmissions. Additionally, concerning scalability, MQTT is supportive of flexible and dynamic connections, which enables easy additions or removal of devices without causing any disruptions to the network.



Table 4.0: Pugh chart for communication technology

Criteria	Weight	Wifi/Baseline	Bluetooth	LORA
Cost-effective	5	0	1	1
Speed/Data rate	5	0	-1	-2
Latency	5	0	-1	-2
Power consumption	5	0	1	1
Security	5	0	0	0
Scalability	4	0	-1	1
<b>Total</b>	<b>29</b>	<b>0</b>	<b>-1</b>	<b>-1</b>

### 3.2.2 Block Diagrams

The Sensor network in Figure 10.0 displays critical components for condition-monitoring induction motors with an ESP32 microcontroller. At the center is the ESP32 microcontroller, surrounded by various sensors and modules. These include a PZEM-004T sensor module (top-left) used to measure voltage and current, a GY-906 camera (middle-left) to measure the temperature of a small region of the motor that is usually the hottest region, an A3144 Hall effect sensor module (top-right) to measure the rotational speed of the motor. Additionally, an AMG8833 thermal sensor (middle-right) monitors temperature through thermal imaging, and an MPU6050 accelerometer (bottom-right) captures vibration data for a detailed analysis of the motor's condition. Together, these components enable comprehensive condition monitoring of induction motors, ensuring efficient and safe operation.

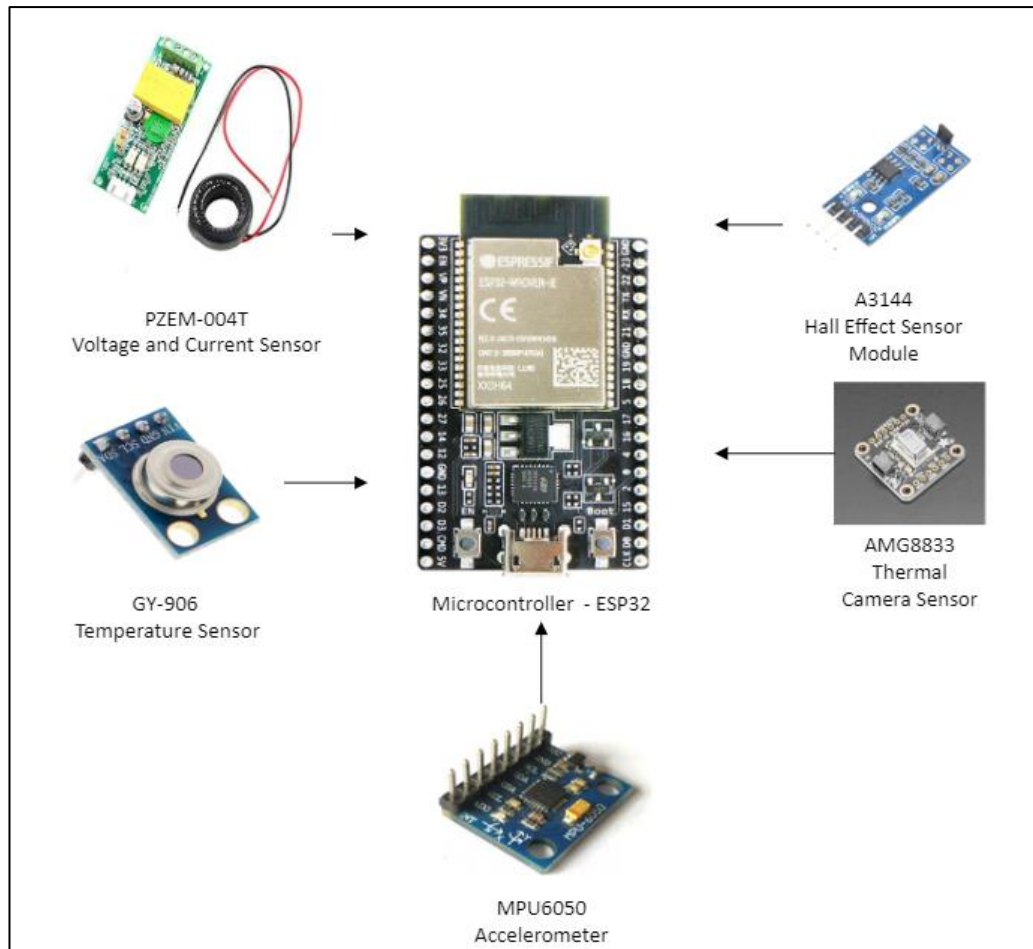


Figure 10.0: Hardware architecture

The block diagram in Figure 11.0 illustrates the data flow in the induction motor condition monitoring system. Sensor data from the ESP32 microcontroller is collected and sent to the Raspberry Pi, which serves as a gateway. The Raspberry Pi then processes these sensor readings. This data is then transmitted using the API, a standardized interface that allows the data to be communicated in a format readable by the centralized database. The stored data is subsequently transformed into a user-friendly interface, enabling real-time monitoring of motor conditions for users using computers and mobile devices. The API ensures seamless data integration between the Raspberry Pi and the database.

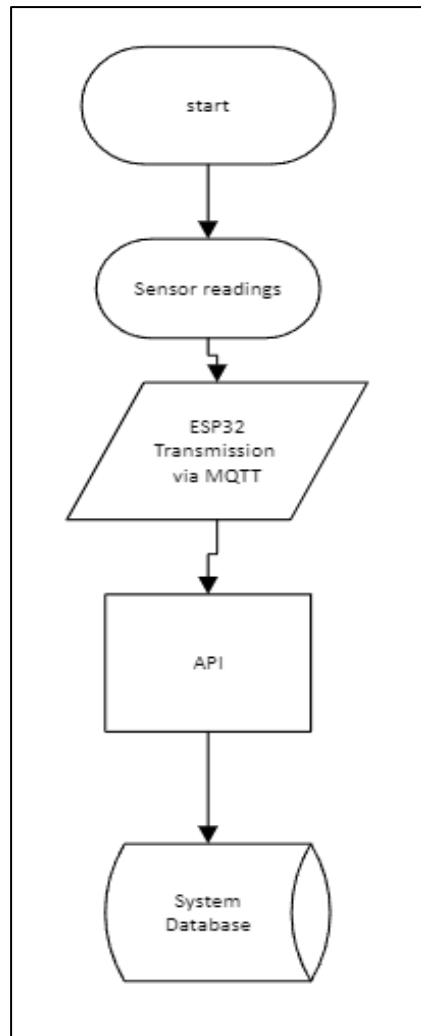


Figure 11.0: Software architecture

The block diagram in Figure 12.0 illustrates the system architecture for condition monitoring of induction motors. As depicted in Figure 10.0, with the ESP32 and various sensors, the sensor network is connected to a Raspberry Pi, which acts as a gateway. This gateway collects sensor readings and sends them to a centralized database. The collected data is then processed and transformed into a user-friendly interface, allowing users to monitor the conditions of the motors in real-time via a computer or mobile device. This setup ensures efficient data collection, storage, and visualization for effective motor condition monitoring.

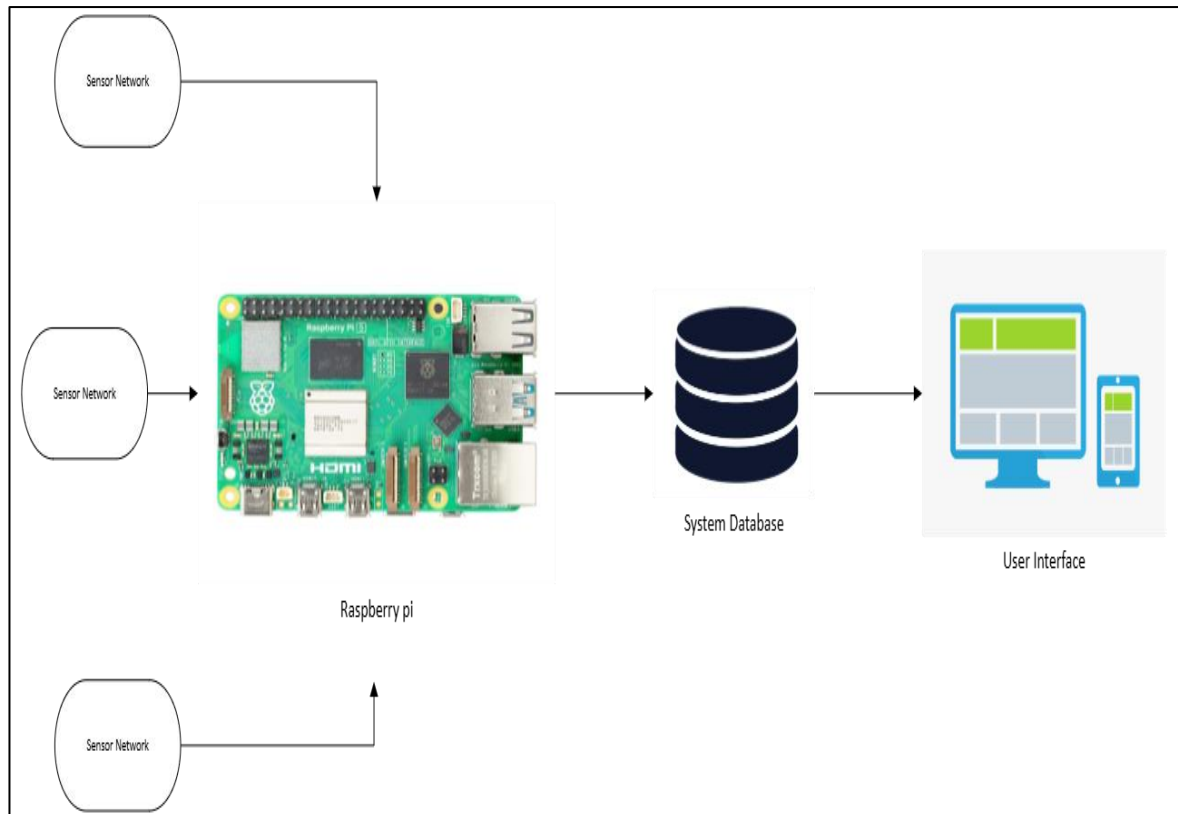

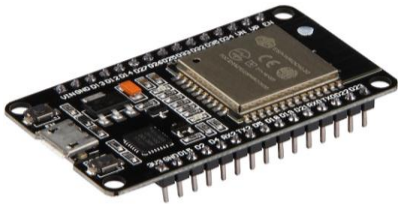
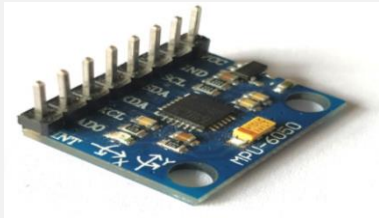
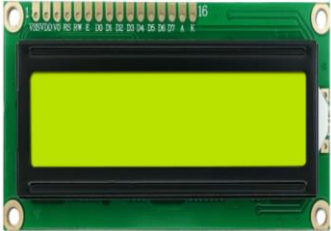





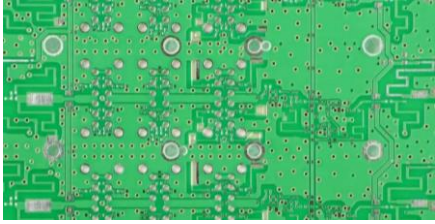
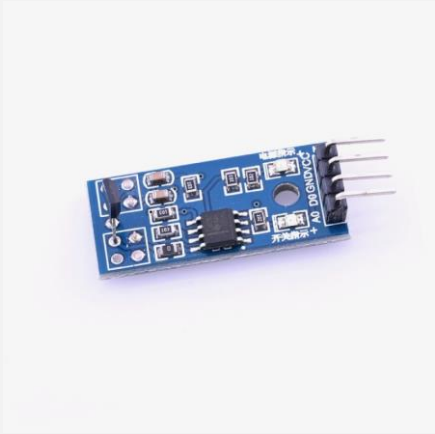
Figure 12.0: System architecture

### 3.2.3 Hardware components

Table 5.0: Hardware components

Component	Picture	Description
<b>Raspberry Pi</b>		<p>Raspberry Pi is a microcontroller unit with many valuable features in IoT applications. Due to its processing ability and large data storage capacity, it will be suitable as the main gateway for data processing and communication with the cloud.</p>

<b>ESP32</b>		<p>This is an energy-efficient, low-cost microcontroller unit. It is equipped with both WIFI and Bluetooth modules. It is easily programmable and can obtain data from all the sensors.</p>
<b>MPU6050</b>  <b>Accelerometer</b>		<p>An accelerometer is used to measure the motor's vibrations of the motor.</p>
<b>LCD</b>		<p>An LCD screen communicates via I2C communication protocol to the microcontroller. This LCD will be used to display the current conditions of the motor.</p>
<b>18650 Li-ion</b>  <b>Battery</b>		<p>Rechargeable Lithium-ion batteries rated at 3.7V</p>
<b>Buzzer</b>		<p>This will indicate when there is a fault in a running machine.</p>

		
<b>LED</b>		A red LED will indicate a fault in the motor.
<b>PCB</b>		The PCB provides mechanical support and electrical connections for electronic components.
<b>A3144</b> <b>Hall Effect</b> <b>Sensor Module</b>		Sensor module that transduces magnetic fields to electrical signals and will help determine the frequency of the motor.
<b>PZEM-004T</b>		A device capable of measuring the high voltage and current received by the motor.


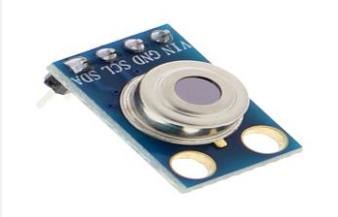
<b>Voltage and Current Sensor</b>		
<b>GY-906 Infrared Thermometer</b>		This device measures temperature of a region using infrared.

Table 6.0: Circuit Calculation

Component	Voltage requirement (V)	Current Requirement (A)
<b>Raspberry Pi</b>	5	<b>1.2</b>
<b>ESP32</b>	3.3	(160-260) m
<b>MPU6050</b>	3-5	4m
<b>LCD</b>	3.3-5	1m
<b>Buzzer</b>	1.5 - 12	20m
<b>LED</b>	1.8 - 3.3	(10 – 30) m
<b>MPU-6050</b>	3.3-5	4m
<b>GY-906 Infrared Thermometer</b>	3.3-5	1.5m
<b>Hall Effect Sensor Module</b>	4.5-5.5	-
<b>PZEM-004T</b>	5	<30m

### 3.3 Circuit Schematic

The electrical system schematic was built using EasyEDA, as shown in Figure 13.0. The schematic built used a battery power system to supply the ESP32. To protect the microprocessor (ESP32) from fluctuating voltages, a voltage regulator, LM7805, with additional electrical components such as capacitors and diodes, produced a smooth voltage output for the microprocessor to operate. The schematic included the sensors being utilized to monitor the condition of the induction motor. The PZEM-004T measures the voltage and current of the induction motor stator. The MPU6050 is an accelerometer that measures the vibrations produced by the induction motor. The GY906 is a precision infrared temperature sensor model that captures the temperature readings of the induction motor in operation. Lastly, the Hall effect sensor is used to measure the speed of the induction motor. The AMG8833 was supposed to be used in this project to capture thermal images, but a FLIR Lepton thermal imaging camera was used due to complications. The difference between the two thermal cameras was that the FLIR had a longer and wider field of view and used more pixels, which meant the thermal imaging data had large sizes.

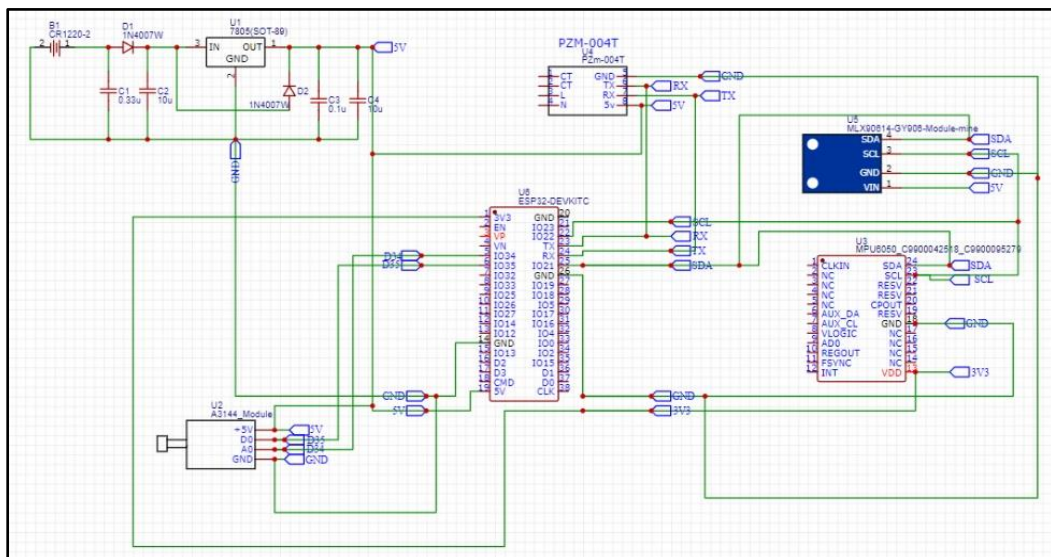


Figure 13.0: Circuit schematic in Easy EDA



### 3.4 Microcontroller and Battery Housing

A CAD model was made of a housing for the monitoring system, as shown in Figure 14.0. This CAD model was designed using SolidWorks. The CAD model was designed to house the circuit boards with the battery power supply, ESP32, and terminal blocks for connection with the sensors. The ESP-32 and the battery power supply were designed to be on the same floor as the CAD model. The width was selected to accommodate the battery power supply, the ESP 32, and the PZEM 004T installed on a perforated board. The height was chosen to reduce the strain of the wires that connected external sensors not installed on the perf board but placed on the motor being monitored, such as the MPU6050 accelerometer, the hall effect sensor, and the GY906 temperature sensor. Stands are designed in solid works, and the sides of the housing are filled with holes to allow easy access to the external sensors and the ESP32.

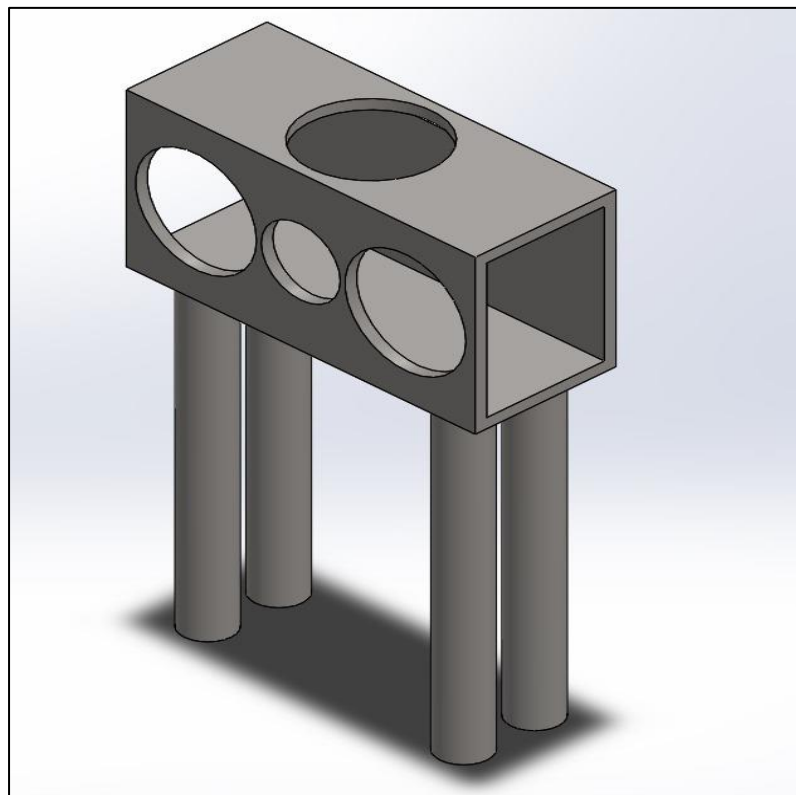


Figure 14.0: 3D model of the microcontroller and battery housing

### 3.5 Motor Selection

To test the condition monitoring system, a suitable motor that was available and whose data could be collected had to be used. Table 7.0 shows the specifications of the physical motor used for the system.

Table 7.0: Motor specifications

Power requirement	240/415 V
Output Power	175 W
Stator Voltage	240/415 V, 3-phase
Rotor Voltage	120/208 V, 3-phase
Full-Load speed	1315 r/min
Full -Load Current	0.48 A
Dimensions(HxWxD)	308 x 291 x440mm (12.1 x 11.5 x 17.3in)
Net Weight	14kg (30.8lb)

### 3.6 MATLAB and Simulink Simulation

A simulation of an induction motor was created on Simulink to replicate data of the induction motor operating with no faults and faults. The healthy induction motor was created using Simulink blocks, and the induction motor parameters were entered using the parameters available in the motor's monitoring datasheet. In Figure 16.0, the full induction motor simulation is seen. Additional simulation blocks were placed in the simulation file to extract data such as the stator current, rotor current, torque induced, rotor speed, rotor frequency, power input, and motor efficiency.

A simulation time of 5 seconds was set with a step size of 0.0005. With the step size and simulation time chosen, each simulation produced 10,000 samples. The healthy motor simulation was run ten times to produce 100,000 samples of healthy motor data. The solver used to run these simulations was Runge-Kutta (ode-4) due to its high level of accuracy and

ability to deal with complex models. The parameters of the induction used were collected from the datasheet available for this motor, and those parameters were inputted into the induction motor Simulink block, as seen in Figure 15.0. The physical motor used for this testing was a three-phase squirrel cage induction motor, and the parameters available in the data sheet in this motor are seen in Table 8.0.

Block Parameters: Induction motor

Asynchronous Machine (mask) (link)

Implements a three-phase asynchronous machine (wound rotor, squirrel cage or double squirrel cage) modeled in a selectable dq reference frame (rotor, stator, or synchronous). Stator and rotor windings are connected in wye to an internal neutral point.

Configuration Parameters Load Flow

Nominal power, voltage (line-line), and frequency [ Pn(VA),Vn(Vrms),fn(Hz) ]: [3001 415 50]

Stator resistance and inductance[ Rs(ohm) Lls(H) ]: [4.989 0.002408]

Rotor resistance and inductance [ Rr'(ohm) Llr'(H) ]: [1.1 0.006]

Mutual inductance Lm (H): 0.1907

Inertia, friction factor, pole pairs [ J(kg.m^2) F(N.m.s) p() ]: [0.02 0.005 1]

Initial conditions  
[slip, th(deg), ia,ib,ic(A), pha,phb,phc(deg)]:  
[0 0 0 0 0 0 0 0]

☐ Simulate saturation Plot

[ i(Arms) ; v(VLL rms)]: 51, 302.9841135, 428.7778367 ; 230, 322, 414, 460, 506, 552, 598, 644, 690]

OK Cancel Help Apply

Figure 15.0: 3D model of the microcontroller and battery housing

Table 8.0: Motor specifications

Power	3kW (4hp) /415 V, 50 Hz
Current	6.3 A
Speed	2860rpm
Power factor	0.87
Locked Rotor Torque / Rated Torque (Tst/TN)	2.2
Locked Rotor Current / Rated Current (Ist/TN)	7.5
Breakdown Torque / Rated Torque	2.3
Power Efficiency	83%
Dimensions (L x W x H)	380mm x 205mm x 245mm

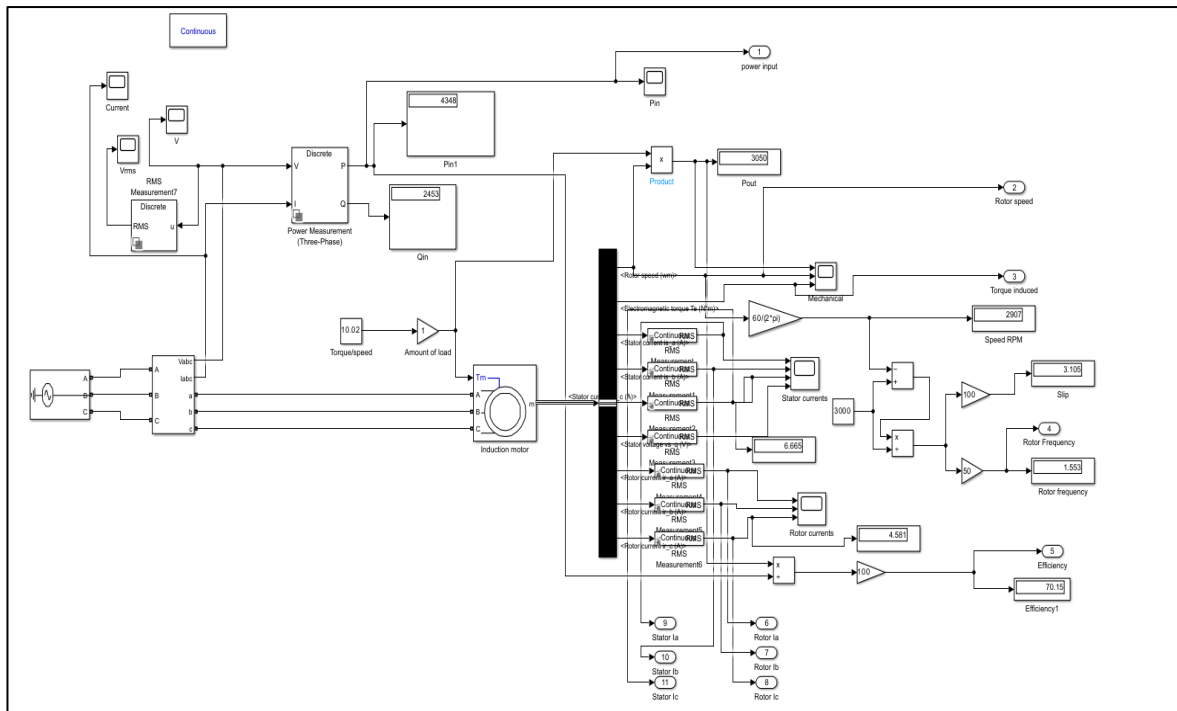


Figure 16.0: MATLAB Simulink model of induction motor

Simulation blocks that induce these faults were added to the healthy motor simulation to generate faulty data. To induce short circuit faults within the induction motor, a three-phase short circuit block was placed on the output of the healthy motor simulation, as seen in Figure 17.0. These three-phase short circuit block caused a sharp increase in the output of the stator currents. The simulation time was kept the same, but the step size was reduced by 0.001, producing 5,000 data samples after running the simulation. Therefore, the simulation was run four times to generate 20,000 samples of faulty data. This allowed for varying the three-phase circuit faults between the three phases and ground and varying the loads to get samples of short circuit fault data. The same process was used for the other faults that were induced.

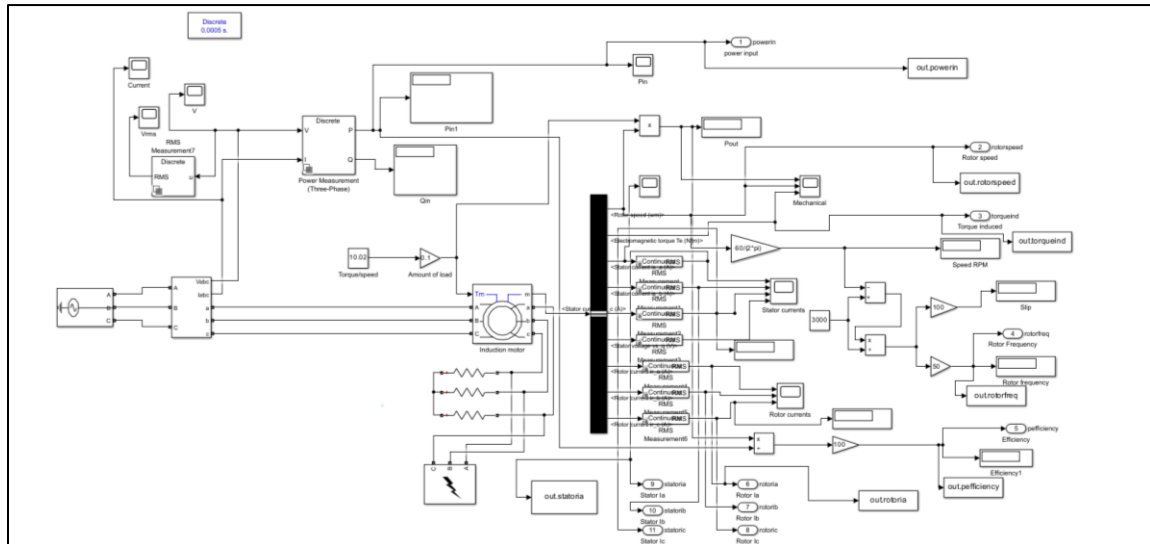


Figure 17.0: Simulink model of induction motor with short circuit fault

The same process was used for the other faults that were induced. Those other faults include an open circuit fault induced by a three-phase circuit breaker, as seen in Figure 18.0, and a broken rotor bar fault caused by an RLC circuit, as seen in Figure 19.0.

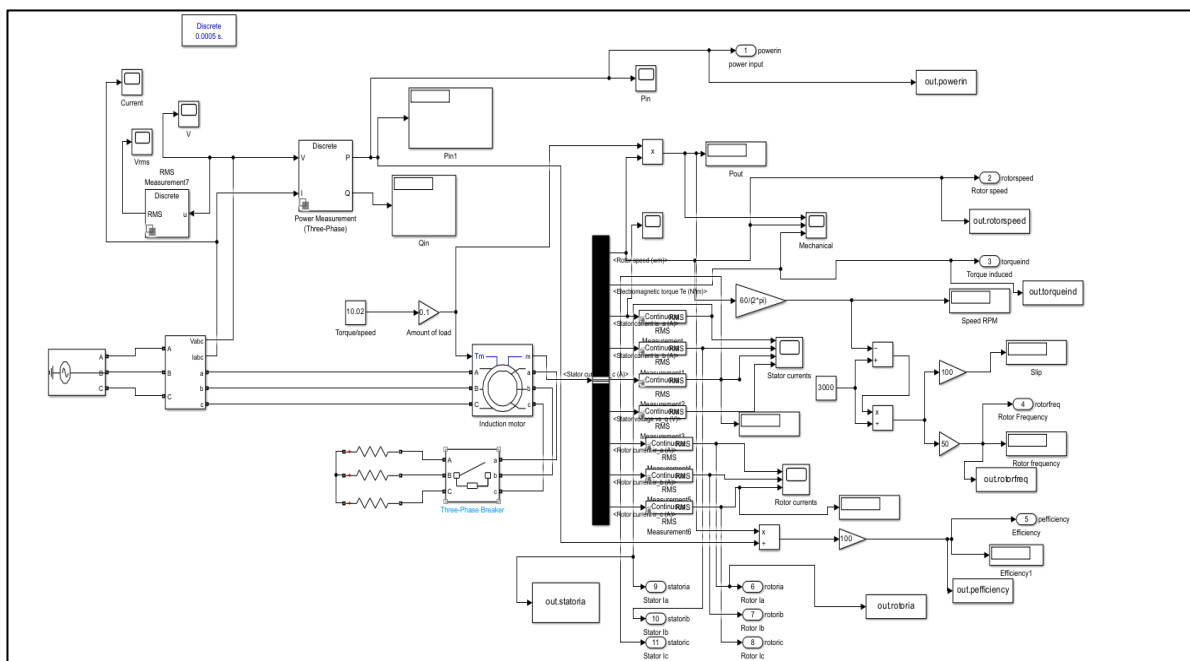


Figure 18.0: Simulink model of induction motor with open circuit fault

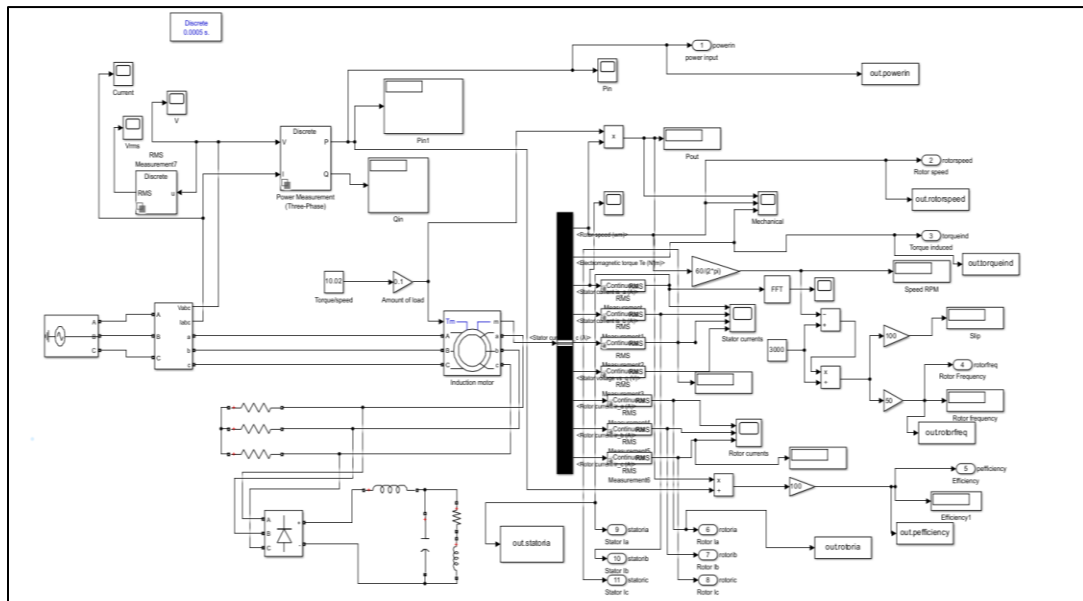


Figure 19.0: Simulink model of induction motor with RLC circuit

The RLC circuit introduced harmonics into the induction motor that mimics a broken rotor bar fault.

## Chapter 4.0: Implementation

This chapter highlights the physical and software implementation of the project. It expounds on how each sub-system was implemented. It also explains the use of data analysis methods to process obtained data.

### 4.1 Electrical System

#### 4.1.1 Perforated Circuit Board

Two perforated circuit boards were created using the electrical schematic design on easyEDA for the system. This consisted of two 7 by 9 cm perforated boards. The perforated board in Figure 20.0 consists of the power regulator to supply power at a steady 5 volts for the ESP32. The next perforated board seen in Figure 20 consists of the ESP32 and multiple terminal pins meant for external power supply to the ESP32, and connections of the pins used to communicate with the network of sensors.

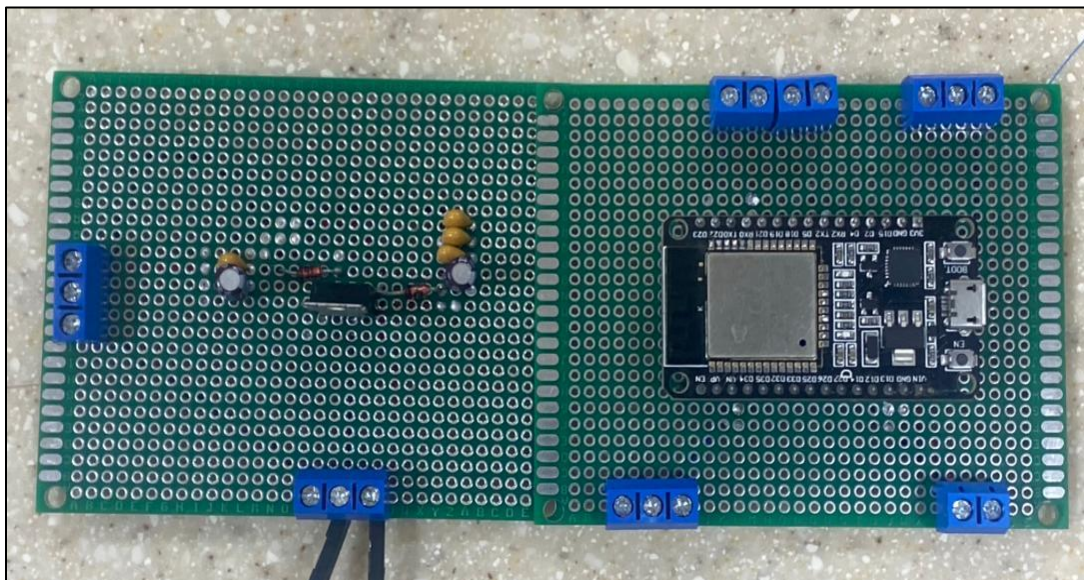


Figure 20.0: Perforated board containing power supply circuitry and terminal blocks for sensor connections.



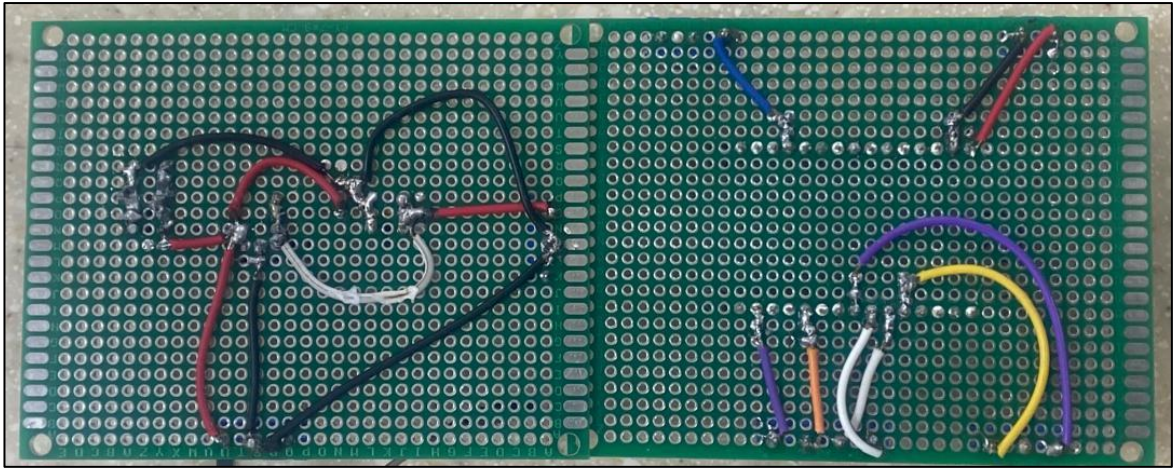


Figure 21.0: Image of the flipped perforated board

#### 4.1.2 Sensor Placement

The PZEM004T was placed within the ESP32 housing and was used to measure the stator current and voltage. Copper wires were used for the connections across the stator windings. The live wire connected across the stator is passed through the current transformer of the PZEM004T with the neutral wire connected to the power supply's ground. The second live wire was connected to one phase of the three-phase power supply, and the second neutral wire was connected to the ground of the three-phase power supply. These four wires were all connected to the pin terminals of the PZEM004T. On the other end of the PZEM004T, it was connected to 5 volts for power and ground, and the RX pin from the PZEM004T was connected to the TX pin of the ESP32 at pin 17, whereas the TX pin of the PZEM004T was connected to the RX pin of the ESP32 at pin 16. The MPU6050 accelerometer was placed on top of the motor using tape as a temporary mounting. The accelerometer was placed on top of the motor parallel to the axis of the shaft because that provided a smooth surface for the accelerometer, which can minimize any extraneous noise captured, as seen in Figure 22.0. This placement also ensures that the vibrational data collected is accurate and can be used to identify motor faults. The hall effect sensor module was placed on the side of the motor closer to the shaft, where an external magnet was placed,



enabling us to calculate the motor's revolutions per minute (RPM). The placement of the hall effect sensor is seen in Figure 23.0. The GY906 infrared thermometer was placed approximately 5 cm from the stator to monitor the operating motor's hottest region. Placing it approximately 5 cm away from the stator allows it to monitor an area with a surface diameter of 8.39 cm, which is relatively close to the diameter of the motor, 10 cm. The placement of this sensor is seen in Figure 24.0.

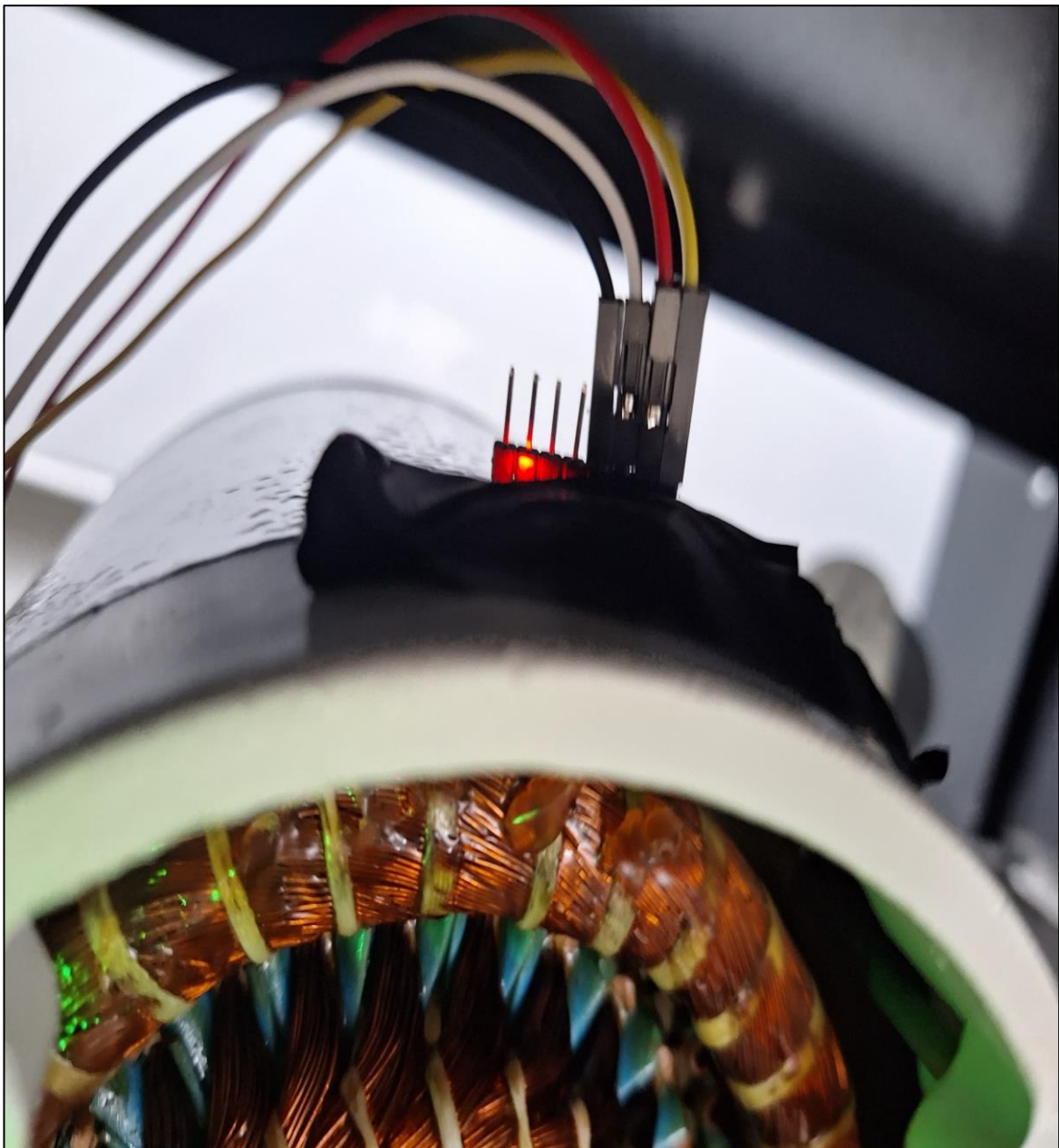


Figure 22.0: Image showing placement of MPU6050 on top of the induction motor

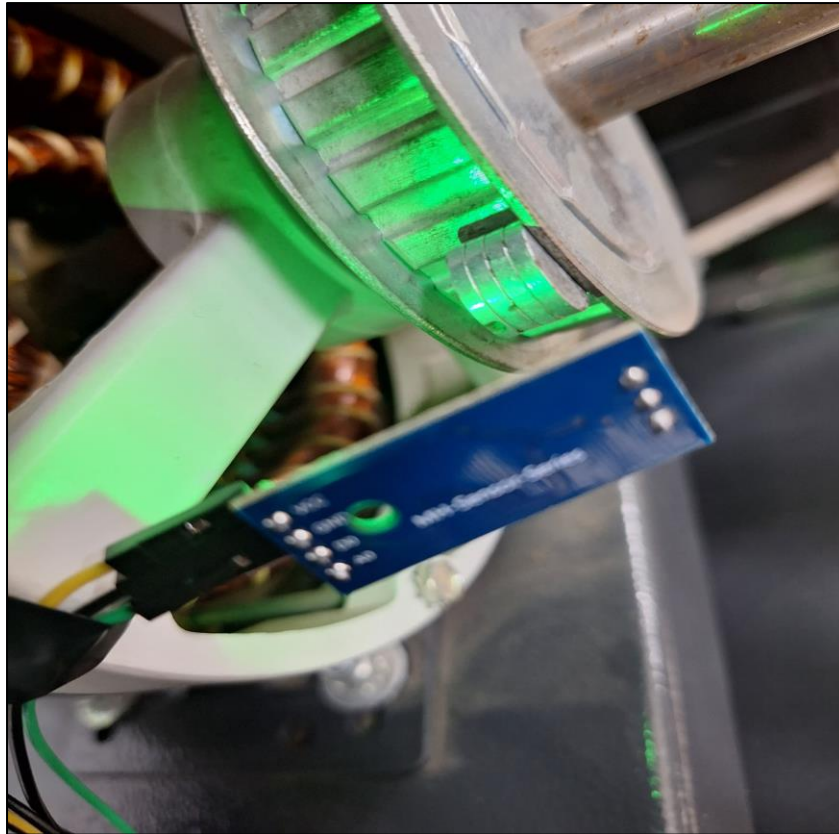


Figure 23.0: Image showing placement of the hall effect sensor module and magnet

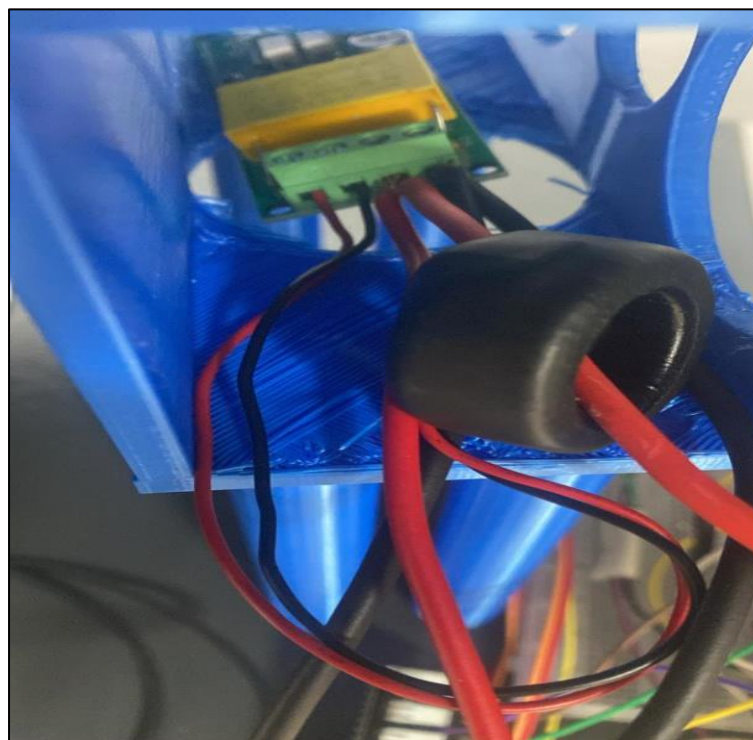


Figure 24.0: Image showing placement of the PZEM-004T in the circuit housing

### **4.1.3 Arduino Code**

To collect data from the various sensors for the ESP32, the Arduino IDE platform was used, where code was implemented to extract data from the sensors for the ESP32. The code included communication with all sensors into one code utilizing the various pins on the ESP32. The libraries for each type of sensor provided a means to read the intended data efficiently. The readings from the hall effect sensor had to be calibrated by measuring the intervals between readings to obtain the motor's RPM. Readings for current, voltage, temperature, and vibration in three planes were more straightforward and were made using the PZEM, GY-906, and MPU6050. The vibration data was collected at 1000 Hz frequency to obtain up to ten harmonics in the motor. The temperature was measured every one second. The current and voltage were also measured every second due to the limitations of the PZEM. Hall effect readings were measured at a five hundred hertz sampling frequency to detect change and calculate the rpm of the motor. The recorded data was then published using MQTT. The main topic all the data went to was motor data. Each parameter had an assigned subtopic to be separately identified. Arduino OTA was integrated into the code to allow the upload of new code over the same network. The Arduino code can be found in Appendix B.

### **4.1.4 Raspberry Pi Setup**

The design involves machine learning on a company's Raspberry Pi. Thus, the fault analysis data is available locally, therefore there was a need to build a simple system that shows the general state of the motors in an industry and gives an alarm when a critical fault occurs. The system designed contains a Raspberry Pi connected to an LCD screen, a buzzer, and an LED. The LCD screen displays valuable information, and the buzzer and LED are powered when a critical fault occurs. A Python script was utilized that reads the data from the analysis and controls all these peripherals.

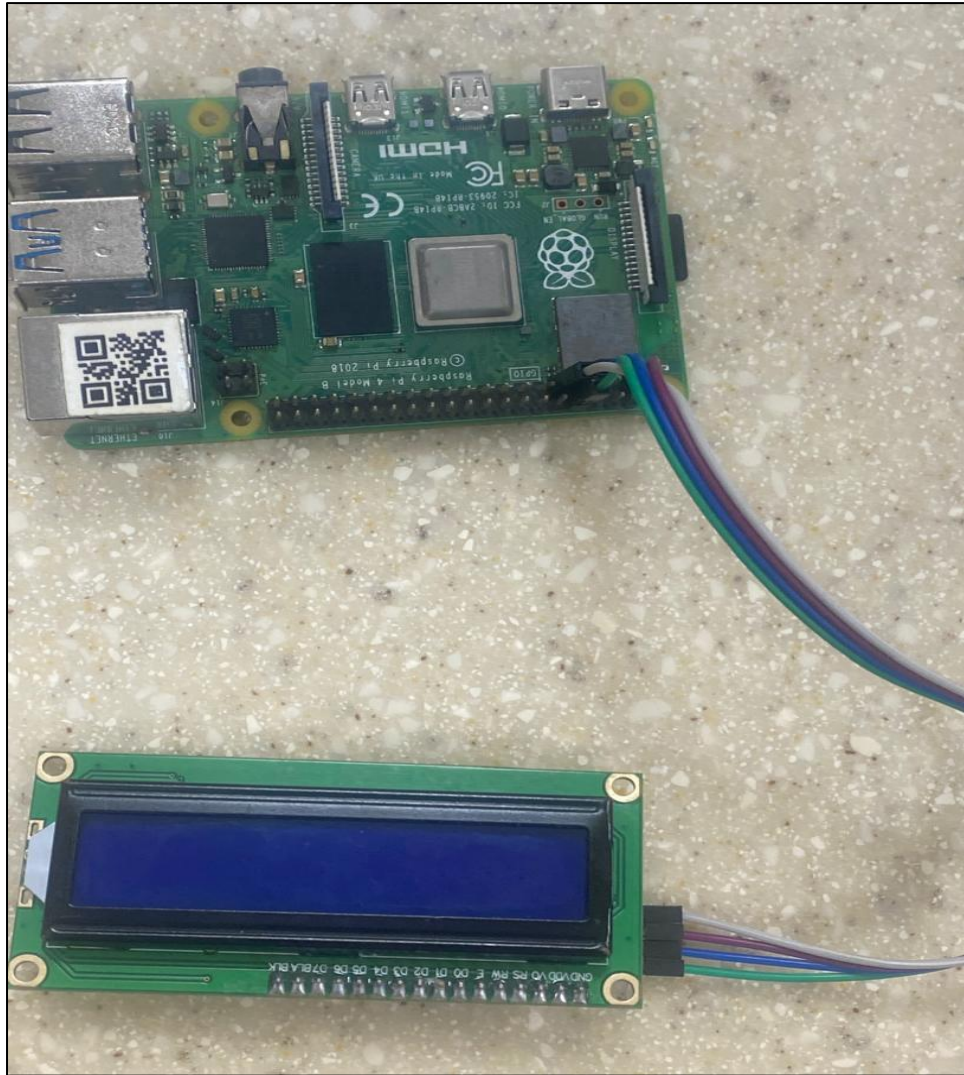


Figure 24.0: Raspberry Pi setup

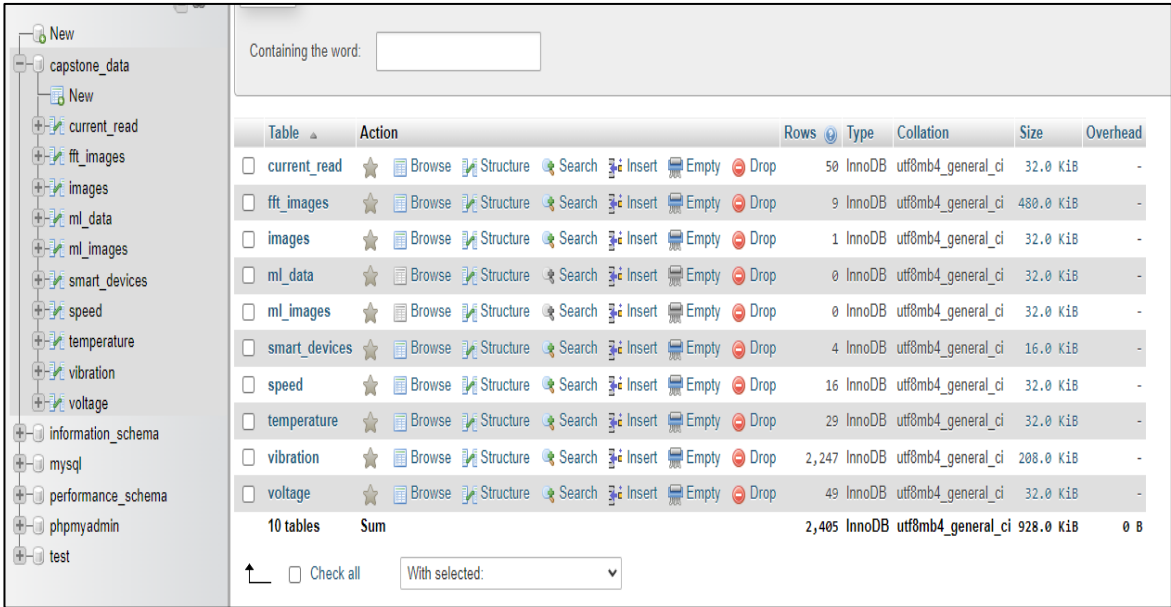
## 4.2 Software System and Implementation

### 4.2.1 Database Design

XAMPP was used to create an SQL database to handle data collected by the system. Due to the system recording data at different sampling rates, the database needed to accept input at varying intervals for several motors to identify each motor's data. Hence, tables were created for each motor parameter being measured. The smart devices table in Figure 25.0 kept the name of each motor circuit and the ID assigned to it. This ID was the reference key for each data entry into the other tables. Figure 26.0 shows the tables in the database.



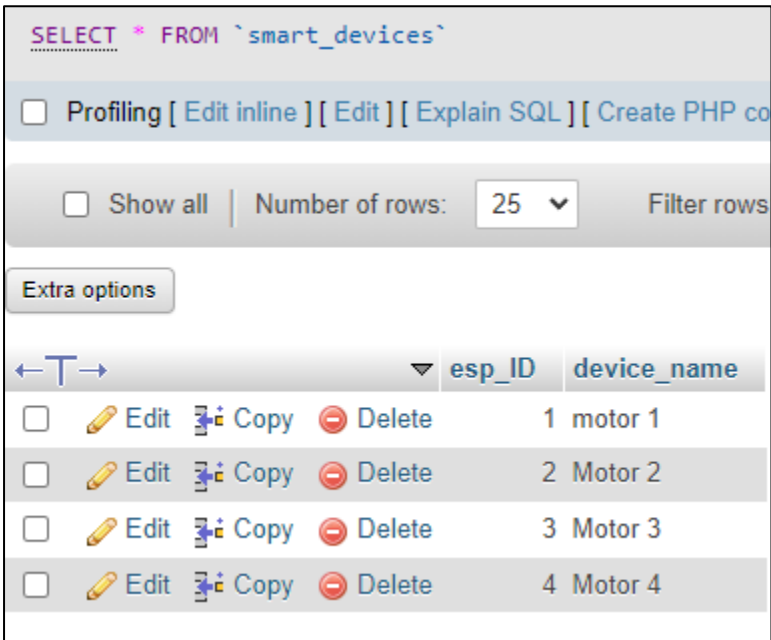
Figure 27.0 shows the structure for the vibration data, and all the other motor parameter tables share the same structure with the esp ID, sensor data, and reading time. The FFT and machine learning image tables also had the same structure with ID, images, and reading time, as seen in Figure 28.0.



The screenshot shows the phpMyAdmin interface for a database named 'capstone\_data'. On the left is a tree view of the database structure, including tables like 'current\_read', 'fft\_images', 'images', 'ml\_data', 'ml\_images', 'smart\_devices', 'speed', 'temperature', 'vibration', and 'voltage'. The main panel displays a table structure overview for 10 tables. Each table has a list of actions (Browse, Structure, Search, Insert, Empty, Drop) and summary statistics (Rows, Type, Collation, Size, Overhead).

Table	Action	Rows	Type	Collation	Size	Overhead
current_read	Browse Structure Search Insert Empty Drop	50	InnoDB	utf8mb4_general_ci	32.0 KiB	-
fft_images	Browse Structure Search Insert Empty Drop	9	InnoDB	utf8mb4_general_ci	480.0 KiB	-
images	Browse Structure Search Insert Empty Drop	1	InnoDB	utf8mb4_general_ci	32.0 KiB	-
ml_data	Browse Structure Search Insert Empty Drop	0	InnoDB	utf8mb4_general_ci	32.0 KiB	-
ml_images	Browse Structure Search Insert Empty Drop	0	InnoDB	utf8mb4_general_ci	32.0 KiB	-
smart_devices	Browse Structure Search Insert Empty Drop	4	InnoDB	utf8mb4_general_ci	16.0 KiB	-
speed	Browse Structure Search Insert Empty Drop	16	InnoDB	utf8mb4_general_ci	32.0 KiB	-
temperature	Browse Structure Search Insert Empty Drop	29	InnoDB	utf8mb4_general_ci	32.0 KiB	-
vibration	Browse Structure Search Insert Empty Drop	2,247	InnoDB	utf8mb4_general_ci	208.0 KiB	-
voltage	Browse Structure Search Insert Empty Drop	49	InnoDB	utf8mb4_general_ci	32.0 KiB	-
10 tables	Sum	2,405	InnoDB	utf8mb4_general_ci	928.0 KiB	0 B

Figure 25.0: Database structure



The screenshot shows the phpMyAdmin interface for the 'smart\_devices' table. At the top, there is a SQL query editor with the query 'SELECT \* FROM `smart\_devices`'. Below the query editor are options for 'Profiling', 'Edit inline', 'Edit', 'Explain SQL', and 'Create PHP code'. There are also options to 'Show all' or 'Number of rows' (set to 25) and a 'Filter rows' button. Below these options is a section for 'Extra options'. The table data is displayed with columns 'esp\_ID' and 'device\_name'. Each row has a checkbox, an 'Edit' button, a 'Copy' button, and a 'Delete' button.

	esp_ID	device_name
<input type="checkbox"/>	1	motor 1
<input type="checkbox"/>	2	Motor 2
<input type="checkbox"/>	3	Motor 3
<input type="checkbox"/>	4	Motor 4

Figure 26.0: Smart devices table

SELECT * FROM `vibration`				
<input type="checkbox"/> Profiling <a href="#">[ Edit inline ]</a> <a href="#">[ Edit ]</a> <a href="#">[ Explain SQL ]</a> <a href="#">[ Create PHP code ]</a>				
1	>	>>	Number of rows:	25
Extra options				
esp_ID	ax	ay	az	reading_time
1	-0.457293	-0.102951	10.7703	2024-07-12 18:14:16
1	-0.136469	0.304064	10.963	2024-07-12 18:14:16
1	0.125696	0.65242	11.2875	2024-07-12 18:14:16
1	0.167594	0.761356	11.467	2024-07-12 18:14:16
1	0.153229	0.752976	11.5796	2024-07-12 18:14:16
1	0.0969652	0.727837	11.6765	2024-07-12 18:14:17
1	-0.00957681	0.620098	11.7507	2024-07-12 18:14:17
1	-0.153229	0.494403	11.7639	2024-07-12 18:14:17
1	-0.282516	0.333991	11.6909	2024-07-12 18:14:17
1	-0.380678	0.223858	11.6598	2024-07-12 18:14:17

Figure 27.0: Vibration data table

esp_ID	reading_time	image
1	2024-07-25 21:44:20	[BLOB - 53.3 KiB]
1	2024-07-25 21:45:04	[BLOB - 45.6 KiB]
1	2024-07-25 21:45:10	[BLOB - 45.6 KiB]
1	2024-07-25 21:45:25	[BLOB - 51.9 KiB]
1	2024-07-25 21:45:30	[BLOB - 45.8 KiB]
1	2024-07-25 21:45:35	[BLOB - 46.3 KiB]
1	2024-07-25 21:45:51	[BLOB - 58.5 KiB]
1	2024-07-25 21:45:54	[BLOB - 52.8 KiB]
1	2024-07-25 21:45:59	[BLOB - 53.1 KiB]

Figure 28.0: Structure of tables with images

### 4.2.2 API

A reliable Application Programming Interface (API) had to be developed to communicate with each table in the database. An API is a set of rules and protocols that allows different software applications to communicate with each other. The API is needed to transmit data to the database in the appropriate format. A configuration file that established the connection to the database based on the database name, username, and password was created, as seen in Figure 29.0. The individual API for each table has a similar structure, with each having the Post method as the main section due to data moving in one direction to the database. The Post method in the API for adding to the vibration and speed table is shown in Figures 30.0 and 31.0.

```
<?php

// Database configuration
$servername = "localhost";
$username = "root";
$password = "";
$dbname = "capstone_data";
$port = 3307; // Use the correct port

// Create connection
$con = new mysqli($servername, $username, $password, $dbname, $port);

// Check connection
if ($con->connect_error) {
    die("Connection failed: " . $con->connect_error);
}

?>
```

Figure 29.0: Configuration code in PHP to connect to the database

```

$method = $_SERVER['REQUEST_METHOD'];

switch ($method) {
    case 'POST':
        $data = json_decode(file_get_contents('php://input'), true);

        if (json_last_error() !== JSON_ERROR_NONE) {
            echo json_encode(['error_message' => 'Invalid JSON input']);
            break;
        }

        $esp_ID = 1; // Set esp_ID to 1
        $S = $data['S'] ?? null;

        if ($S === null) {
            echo json_encode(['error_message' => 'Missing required parameter: S']);
            break;
        }

        // Prepare the SQL statement to insert sensor readings
        $stmt = $con->prepare('INSERT INTO speed (reading_time, esp_ID, S) VALUES (CURRENT_TIMESTAMP, ?, ?)');

        // Bind parameters to the prepared statement
        $stmt->bind_param("ii", $esp_ID, $S); // Use "d" for double type and "i" for integer type

        // Execute the prepared statement
        if ($stmt->execute()) {
            echo json_encode(['success_message' => 'Sensor readings added successfully']);
        } else {

```

Figure 30.0: API to post to the speed table

```

$method = $_SERVER['REQUEST_METHOD'];

switch ($method) {
    case 'POST':
        $data = json_decode(file_get_contents('php://input'), true);
        if (json_last_error() !== JSON_ERROR_NONE) {
            echo json_encode(['error_message' => 'Invalid JSON input']);
            break;
        }

        $esp_ID = 1; // Set esp_ID to 1
        $ax = $data['ax'] ?? null;
        $ay = $data['ay'] ?? null;
        $az = $data['az'] ?? null;

        if ($ax === null || $ay === null || $az === null) {
            echo json_encode(['error_message' => 'Missing required parameters']);
            break;
        }

        // Prepare the SQL statement to insert sensor readings
        $stmt = $con->prepare('INSERT INTO vibration (reading_time, esp_ID, ax, ay, az) VALUES (CURRENT_TIMESTAMP, ?, ?, ?, ?)');

        // Bind parameters to the prepared statement
        $stmt->bind_param("idd", $esp_ID, $ax, $ay, $az); // Use "d" for double type and "i" for integer type

        // Execute the prepared statement
        if ($stmt->execute()) {
            echo json_encode(['success_message' => 'Sensor readings added successfully']);
        } else {

```



Figure 31.0: API to post to the Vibration table

## **4.3 Data Analysis Methods**

### **4.3.1 Fast Fourier Transform**

To correctly assess the health and performance of the induction motor being monitored, some analysis of the data collected from the accelerometer and the voltage and current readings. After data collection, Fast Fourier transform (FFT) is applied to the raw vibrational data to transform them from the time domain to the frequency domain. Key frequencies, such as the fundamental frequencies of the induction motor, are identified, and the magnitude produced by those frequencies is analyzed as those frequencies, along with their sidebands, are associated with common faults in an induction motor. Statistical analysis is performed on the FFT results, which are compared to that of the FFT of the healthy motors. Based on the results of that comparison, the motor will be indicated as healthy or faulty.

### **4.3.2 Machine Learning**

The machine learning implemented in this code had two goals. The first was to determine when different types of faults occur in the induction motor. The second goal is to predict when multiple faults co-occur. A separate model had to be created to obtain the machine learning objectives. The best model for each task was then saved and stored on the Raspberry Pi, which would then periodically be used to make predictions on the state of the motor.

#### **4.3.2.1 Dataset used**

The data for machine learning is pivotal in training the best model. For this project, the data required for maximum accuracy had several fields corresponding to the many potential states related to motor health. The motor may be healthy or have several types of

faults, such as short circuit faults, open circuit faults, and misalignment faults. The motor used in this research could not be intentionally damaged; therefore, the only type of data collected by the designed system was for normal operation. Since this was insufficient to train the model, data from other sources had to be used. The first data source is from the MATLAB Simulink simulation of an induction motor designed as stated before. The simulations provided data for four possible states. These are normal operation, open circuit faults, short circuit faults, and broken rotor bar faults. The motor specifications used in the simulation were similar to the physical motor used for the study. This ensured the model trained with the data would work with the physical motor. The second data source was obtained from the machinery fault dataset on Kaggle. The motor states considered were regular operation, horizontal misalignment, vertical misalignment, and motor imbalance. Over two hundred and fifty thousand data points were collected and used in training the model.

#### **4.3.2.2 Machine Learning Techniques**

There were two categories of techniques used for implementing machine learning. The literature from related research showed that Artificial Neural Networks (ANN) was the most accurate at classifying induction motor faults. Thus, ANN is the primary model used to identify fault occurrences. The ANN model we built was compared to the models generated by SVM, Gradient Boosting, and Random Forest techniques as a control measure.

Several machine learning techniques were used to make predictions on when the motor would be at fault. They are Random Forest Regressor that produces decision trees during training and outputs the mean prediction of individual trees, Long Short-Term Memory (LSTM) Network, which is a recurrent neural network (RNN) that can make predictions using long term dependencies, and Support Vector Regressor (SVR).

Appendix D shows the code used to perform machine learning on the collected data.

#### **4.3.2.3 Imported Dependencies**

To run the machine learning model, the essential libraries and toolboxes needed were:

1. Pandas: Provides data structures and analysis tools.
2. Numpy: Provides support for large arrays and the mathematical function to operate on them.
3. sklearn.preprocessing: Used for feature scaling to normalize the data.
4. sklearn.model\_selection: Provides functions for splitting the Dataset, performing cross-validation, and hyperparameter tuning.
5. sklearn.metrics: Provides functions to evaluate the performance of machine learning models.
6. sklearn.ensemble: Provides ensemble learning methods to improve the performance of the model.
7. sklearn.SVM (SVR): Support Vector Regression, a type of Support Vector Machine used for regression tasks.
8. sklearn.feature\_selection: Used for selecting the best features based on statistical tests.
9. imblearn.over\_sampling: Used for handling imbalanced datasets by generating synthetic samples.
10. TensorFlow.keras.models: Used to create a sequential neural network model.
11. TensorFlow.keras.layers: Provides layers for building the neural network, including LSTM layers for time series data and Dense layers for fully connected neural networks.

12. shap: Used for model interpretability by explaining the output of machine learning models.

13. matplotlib.pyplot: Used for plotting graphs and visualizations.

14. seaborn: A data visualization library based on matplotlib that provides a high-level interface for drawing attractive and informative statistical graphics.

#### 4.4 Website Design

A website was created to give users access to the data collected from each motor. The website has four pages to make it simple and appealing to the user. The first page, Figure 32.0, is a home page that introduces the user to the project's purpose and explains why condition monitoring is essential. Figures 33a.0 and 33b.0 show the Motor Data page, where recently recorded motor data using the sensors is extracted from the database and exhibited to the user. The user is given an option to select which motor data they would like to view. This helps them to examine if the system is working for each motor. The third page, Figure 34.0, is for Data Analysis, and it shows plots obtained from FFT and machine learning alongside the interpretation of the results from the data collected for each motor. The last page, labeled Suggestion in Figure 35.0, displays the motors at fault that need maintenance.

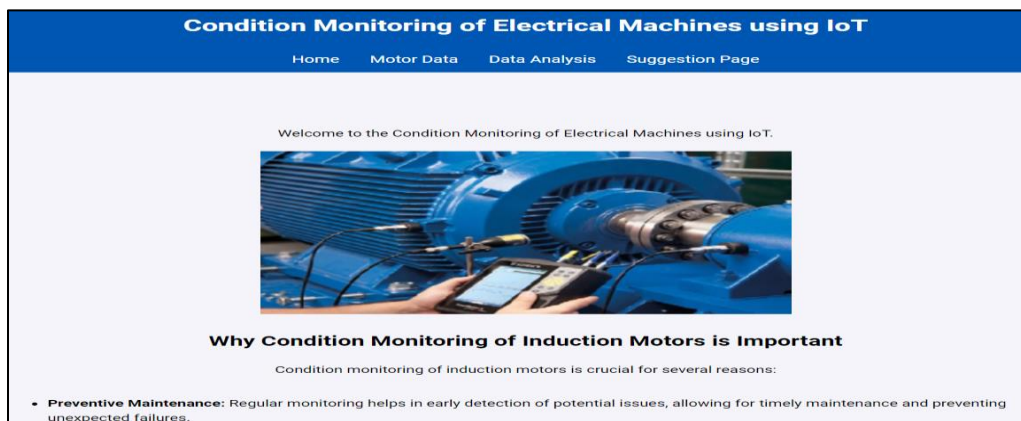


Figure 32.0: Home Page

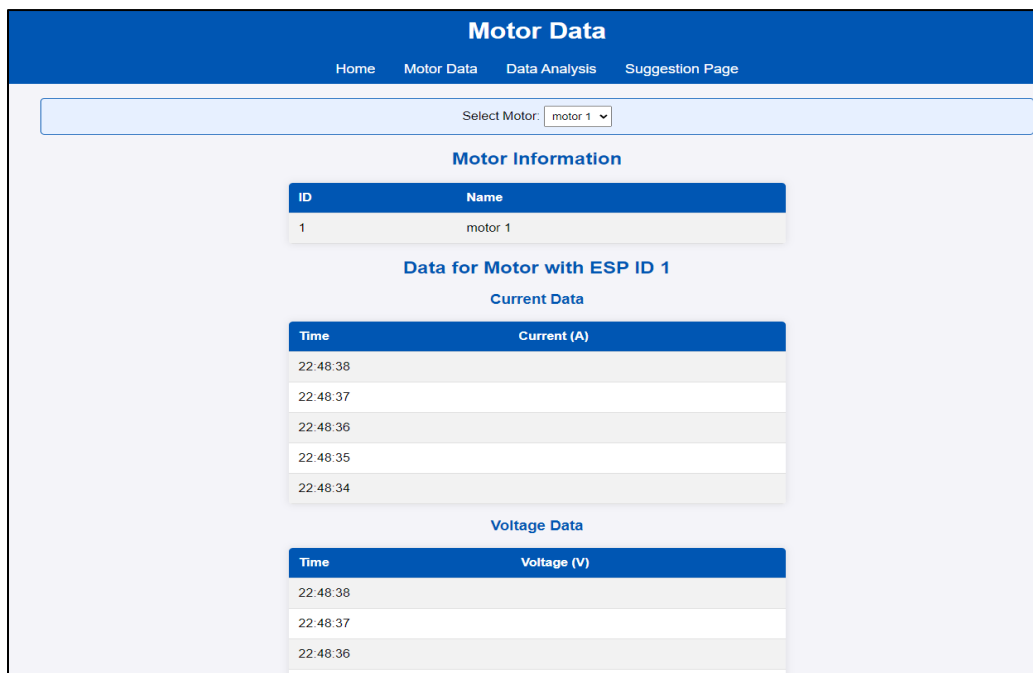


Figure 33a.0: Motor Data Page

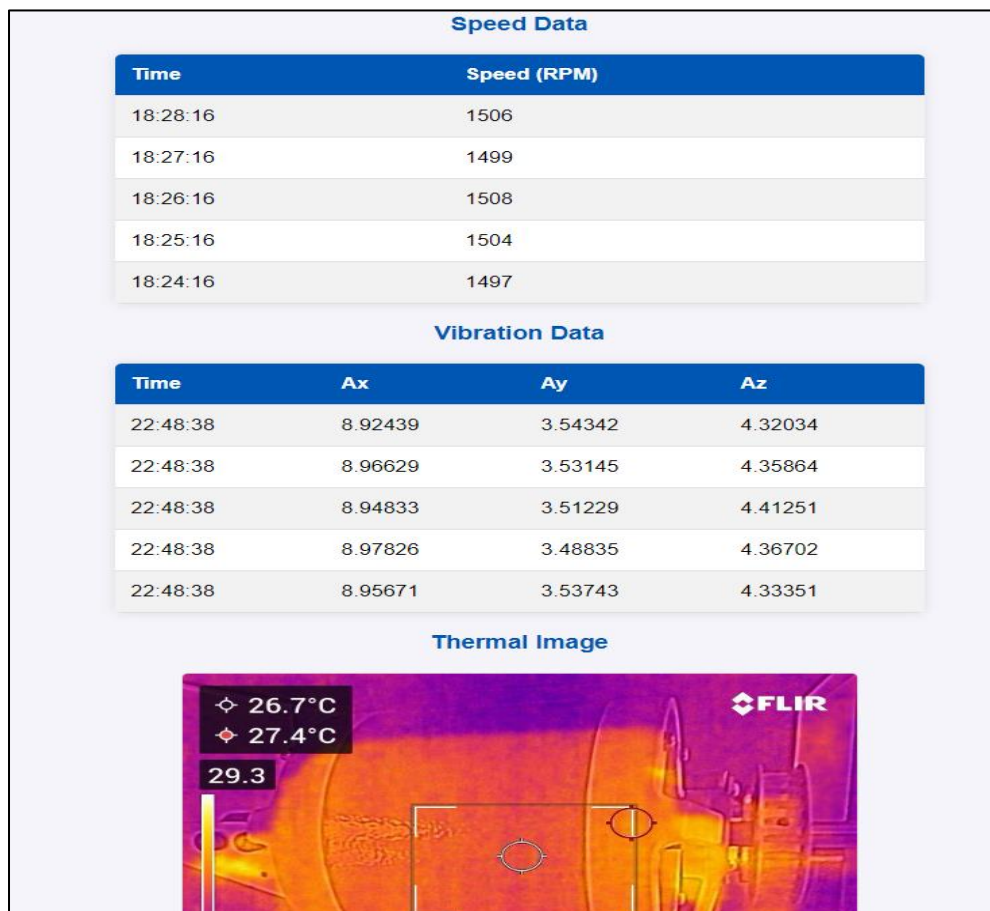


Figure 33b.0: Motor Data Page

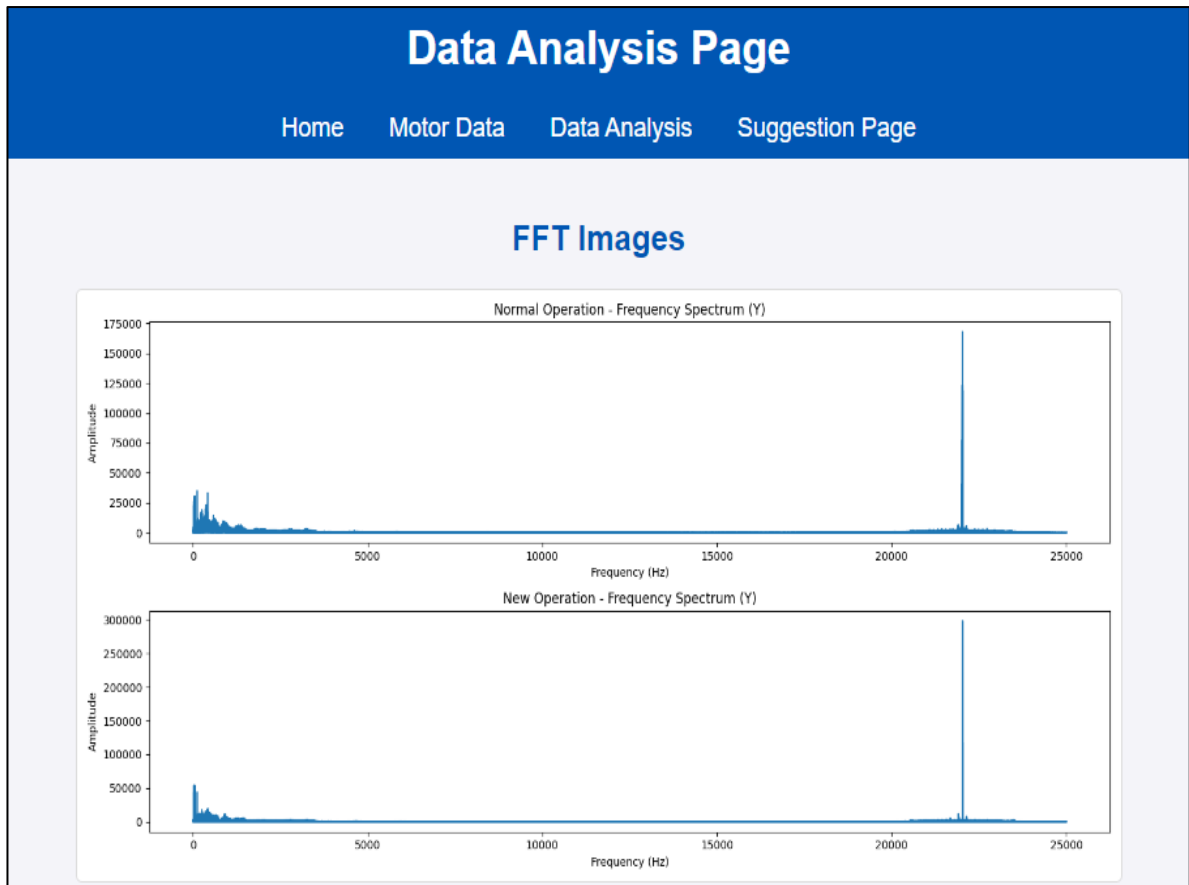


Figure 34.0: Data Analysis Page

The screenshot displays the 'Suggestion Page' with a navigation bar containing 'Home', 'Motor Data', 'Data Analysis', and 'Suggestion Page'. Below the navigation bar is a 'Select Motor:' dropdown menu with 'Motor 1' selected and a 'Submit' button. Below this is a white box containing the following text:

**Motor ID: 1**

The state of Motor 1 is currently healthy according to the analysis conducted using Artificial Neural Networks (ANN).

However, predictive maintenance analysis indicates that Motor 1 is expected to experience a fault within the next 3 months.

It is recommended to schedule a maintenance check within this period to prevent any potential downtime or damage.

Further analysis and maintenance suggestions are recommended based on these results to ensure optimal motor performance and longevity.

Figure 35.0: Suggestion Page

## Chapter 5.0: Results and Discussion

This chapter discusses the results of the physical implementation of our condition monitoring system. It discusses the Simulink simulation results, and the Dataset collected, the data processing of vibrational data, and the website results from implementing the machine learning model.

### 5.1 MATLAB and Simulink

The results of the healthy motor are shown using a scope block in Simulink. The multiple scope blocks show different parameters resulting from running the induction motor simulation under healthy conditions. The scope showcased in Figure 36.0 showcases the torque and speed in rad per second at full load during normal conditions. Figure 37.0 showcases the stator currents of the motor under these same conditions. After the rotor speed, torque, and current stabilize, the figures result in a graph with minimal oscillations.

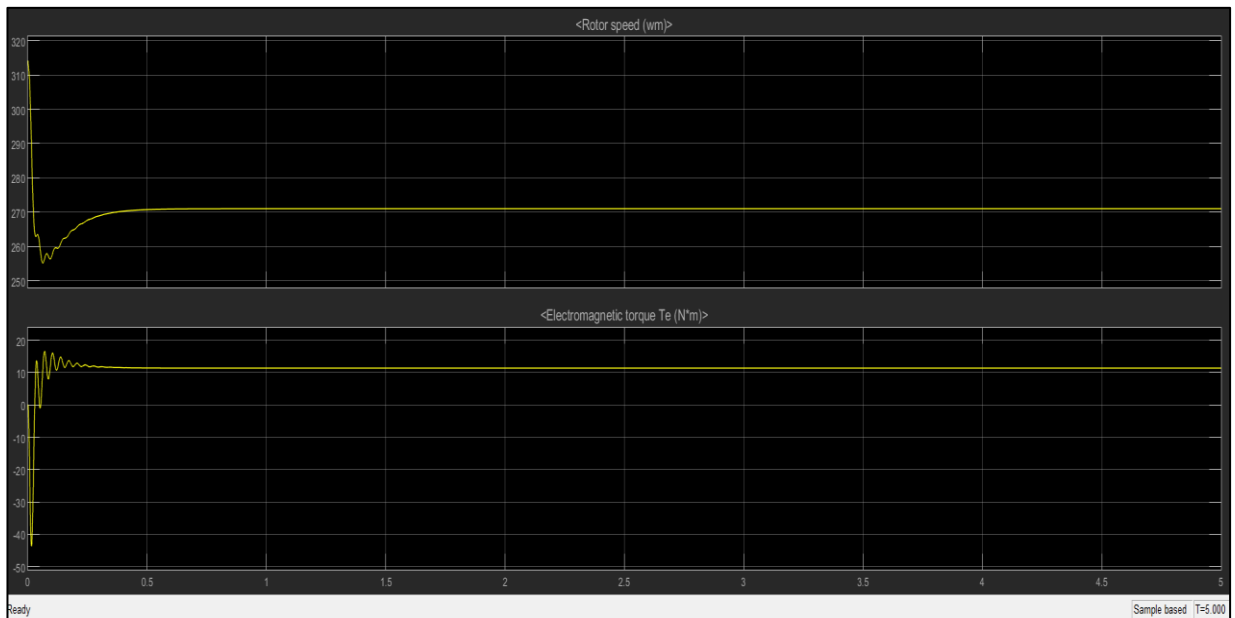


Figure 36.0: Torque and speed of induction motor under full load and normal conditions

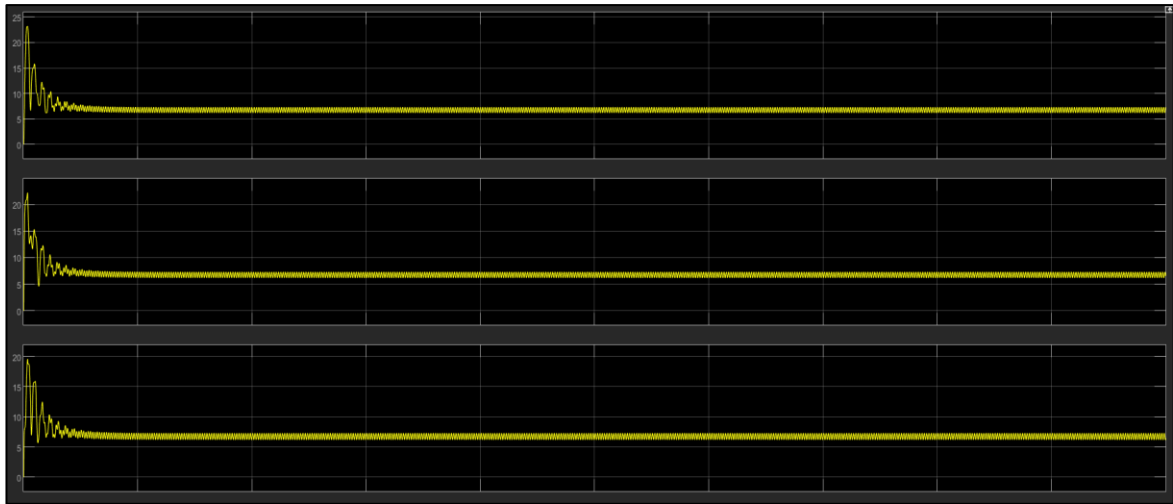


Figure 37.0: Stator currents of an induction motor under full load and normal conditions

After the short circuit fault was implemented in the simulation, the results of the torque and speed are seen in Figure 38.0, and the stator currents are shown in Figure 39.0. The results in Figure 39.0 shows a significant increase in the current readings of the stator, which can cause damage to the motor. The torque and speed of the motor also showcase violent oscillations due to the short circuit fault.

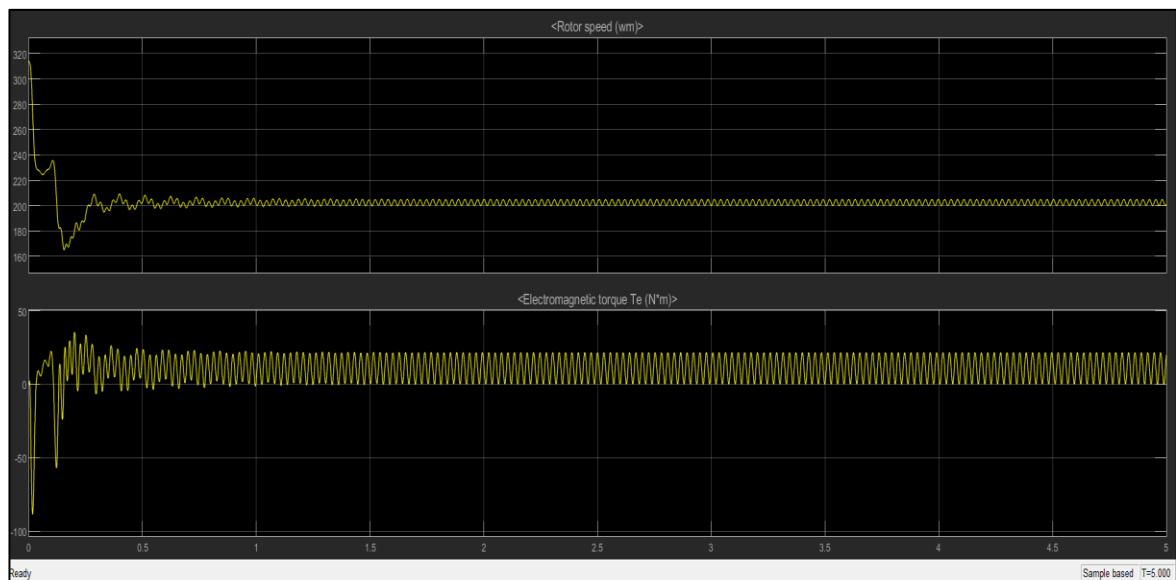


Figure 38.0: Torque and speed of induction motor under full load and short circuit fault



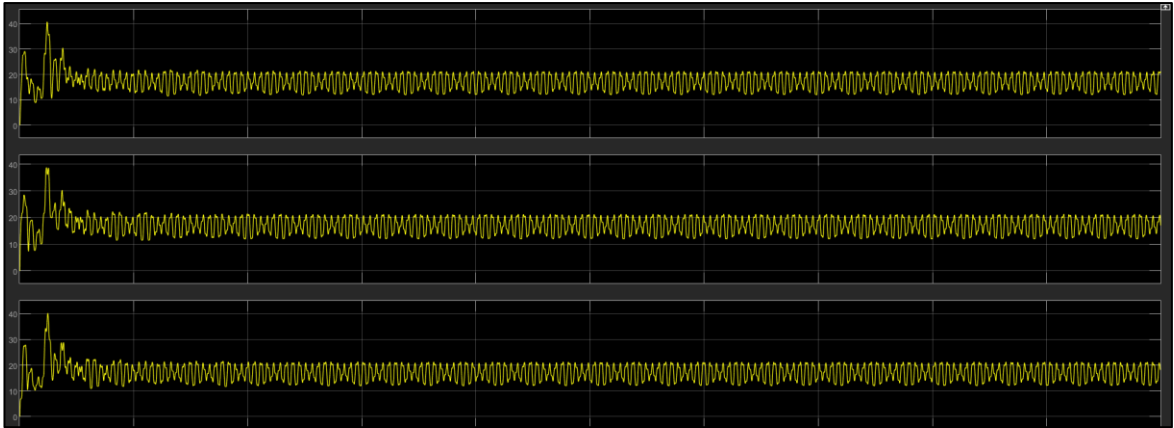


Figure 39.0: Torque and speed of induction motor under full load and short circuit fault

Next, an open circuit fault was implemented and run in the simulation. The results of the torque and speed are seen in Figure 40.0 and the results for the stator current are shown in Figure 41.0. The results in Figure 41.0 show a reduction in the current readings of the stator as expected from an open circuit fault. The lack of current is due to the open circuit formed, which caused oscillations in both the speed and torque of the motor, as seen in Figure 40.0, as well as a reduction of the speed of the motor.

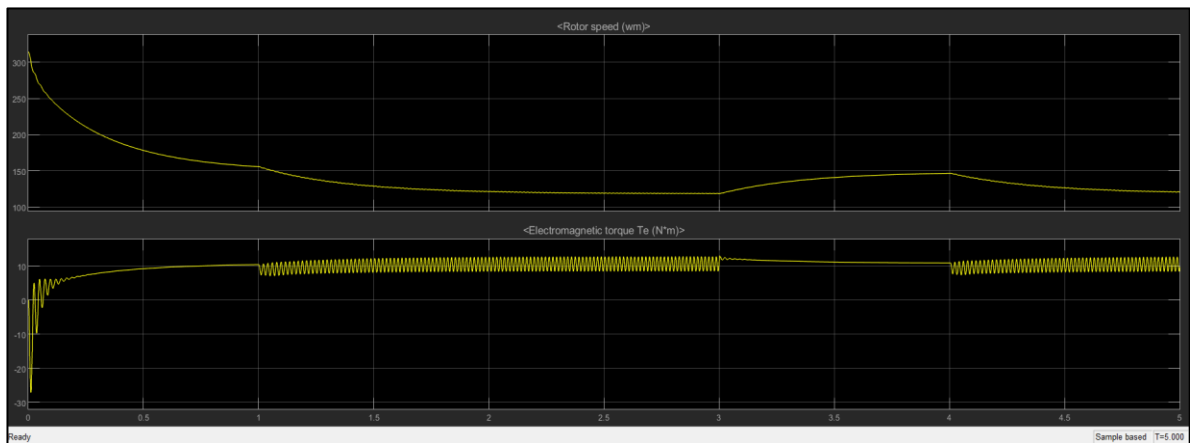


Figure 40.0: Torque and induction motor speed under full load and open circuit fault

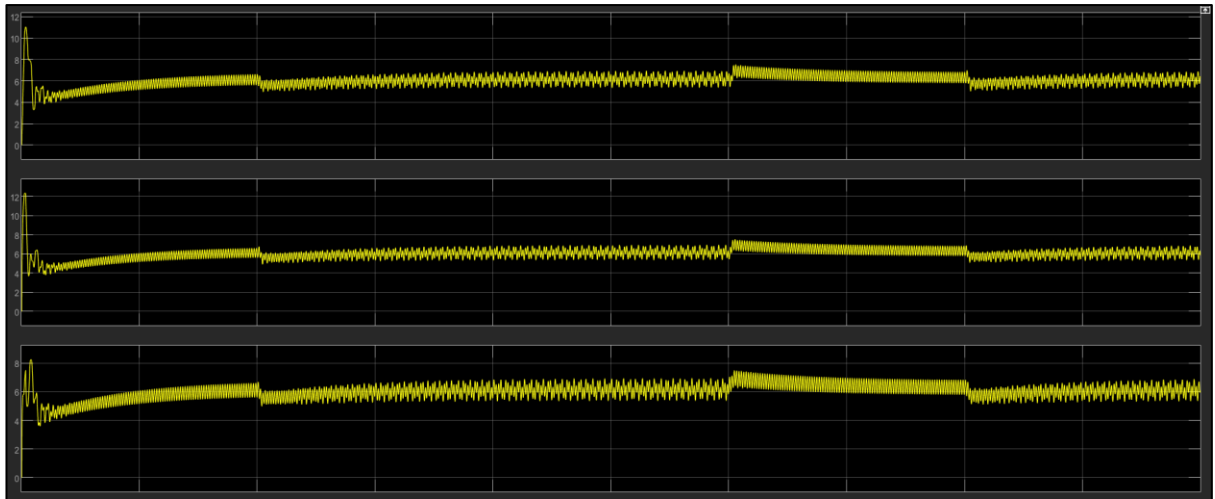


Figure 41.0: Stator currents of induction motor under full load and open circuit fault

The last fault induced was the broken rotor bar fault. Introducing harmonics into the stator currents induced the broken rotor bar fault. The effect on the stator currents is seen in Figure 43.0, where the sinusoidal currents produced are distorted, and the impact on the speed and torque shows some unwanted oscillations in Figure 42.0.

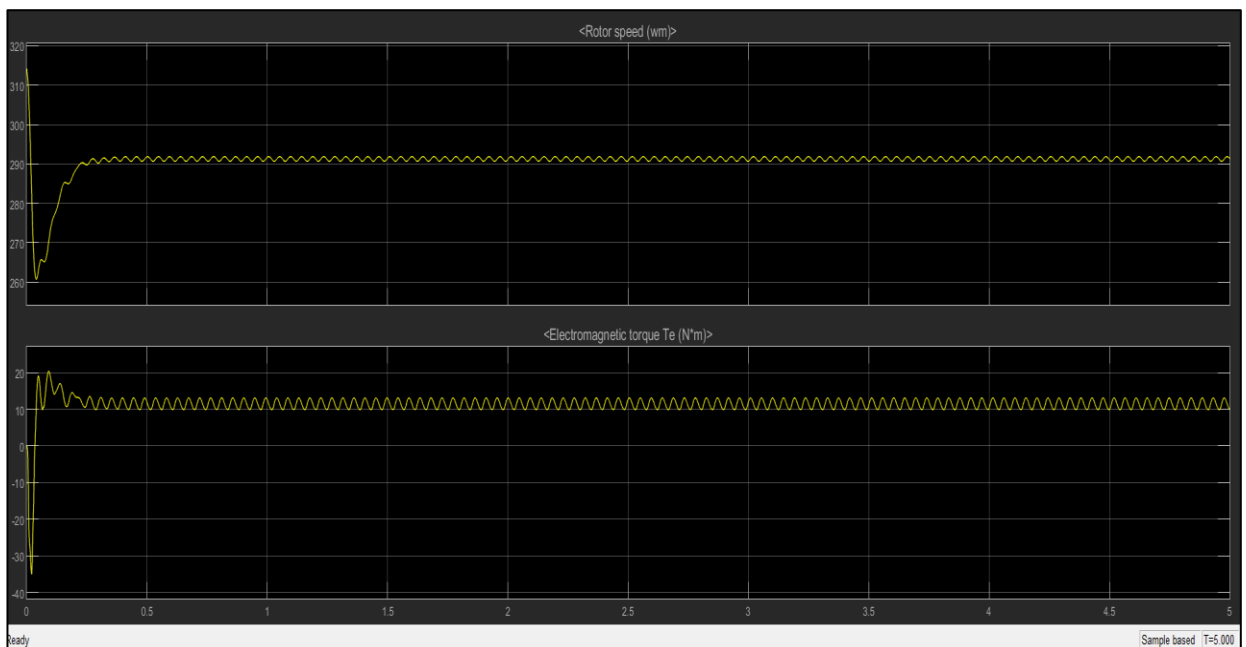


Figure 42.0: Torque and speed of induction motor under full load and Broken rotor bar fault



Figure 43.0: Stator currents of an induction motor under full load and Broken rotor bar fault

## 5.2 Electrical System

The electrical system consisting of the ESP32 and perforated boards with the power supply and terminal blocks for connection with the sensors were placed by the motor at an optimal distance. Jumper wires were connected and passed through the holes to the sensors, and the entire system was activated. A python script in Appendix C was then used to subscribe to each sensor's parameter data and post to the API, which sent the data to the database. The Python script was run on the Raspberry Pi, the server that could subscribe and post data from multiple devices' multi-sensor modules. Figure 44.0 shows the output of the terminal when the posting was done.

```
Received message: { "AccelX":0.183156431, "AccelY":0.548272192, "AccelZ":11.6717329 } on topic motor_data/vibration_data
Sending data to URL: http://192.168.247.198/Capstone/3wObjectsRest/vibration_api.php with payload: {'ax': 0.183156431, 'ay': 0.548272192, 'az': 11.6717329}
API Response: 200, Response Text: {"success_message":"Sensor readings added successfully"}
Received message: {"RPM":0} on topic motor_data/speed
Unknown parameter name
Received message: {"Voltage":230.5} on topic motor_data/voltage
Sending data to URL: http://192.168.247.198/Capstone/3wObjectsRest/voltage.php with payload: {'V': 230.5}
API Response: 200, Response Text: {"success_message":"Sensor readings added successfully"}
Received message: {"Current":0.035999998} on topic motor_data/current
Sending data to URL: http://192.168.247.198/Capstone/3wObjectsRest/current_read.php with payload: {'C': 0.035999998}
API Response: 200, Response Text: {"success_message":"Sensor readings added successfully"}
Received message: {"Temperature_C":24.56529427} on topic motor_data/temperature
Sending data to URL: http://192.168.247.198/Capstone/3wObjectsRest/temperature.php with payload: {'temp': 24.56529427}
API Response: 200, Response Text: {"success_message":"Sensor readings added successfully"}
Received message: {"AccelX":-0.502782345,"AccelY":0.721851826,"AccelZ":11.37604904} on topic motor_data/vibration_data
Sending data to URL: http://192.168.247.198/Capstone/3wObjectsRest/vibration_api.php with payload: {'ax': -0.502782345, 'ay': 0.721851826, 'az': 11.37604904}
API Response: 200, Response Text: {"success_message":"Sensor readings added successfully"}
```

Figure 44.0: Output of Raspberry Pi python script subscribing and posting motor data.

## **5.5 Data Processing**

### **5.5.1 Fast Fourier Transforms**

Due to the system being expected to detect faults as quickly as possible when they occur, a data processing system was implemented that quickly detected faults. Since most faults can be identified by monitoring the vibration, a script that performs calculations on incoming data to determine its frequency components was implemented. The frequency components of the new data were then compared to the frequency components of the normally operating motor. The script also considers the motor's temperature and compares it to a set value. This script was run on Python, and the Dataset used to test it was Kaggle. The output showed that it could identify faults when they occur by performing T-test and KS-test. The tests compared new FFT data collected to the baseline healthy FFT data. The T-test compares the mean amplitudes of the FFT results between the new and baseline data with a confidence level of 95%. If a statistically significant difference is found, the script will print that out. The next KS test compares the distribution of the FFT results of the baseline and the new data and checks if there is a statistical difference between them. Both tests used a confidence level of 95% and a significance level of 0.05. Both tests produce p-values; if this p-value is less than the confidence level, it means a significant difference between the two datasets. The snippet of the code performing this action is seen in Figure 45.0.

This script was on the Raspberry Pi, and it could identify faults when they occurred. The results were printed on an LCD screen. The Raspberry Pi setup is shown in Figure 24. The results of the FFT script are shown in Figures 46.0 and 47.0.

```

t_stat, p_value = ttest_ind(yf_normal, yf_new, equal_var=False)
ks_stat, ks_p_value = ks_2samp(yf_normal, yf_new)

if p_value < 0.05:
    print(f"Significant difference detected in FFT amplitudes for column {axis_labels[col]} (t-test p-value: {p_value}).")
else:
    print(f"No significant difference detected for column {axis_labels[col]} (t-test p-value: {p_value}).")

if ks_p_value < 0.05:
    print(f"Significant difference detected in FFT distributions for column {axis_labels[col]} (KS test p-value: {ks_p_value}).")
else:
    print(f"No significant difference detected in FFT distributions for column {axis_labels[col]} (KS test p-value: {ks_p_value}).")

```

Figure 45.0: Code that performs the T-test and KS test

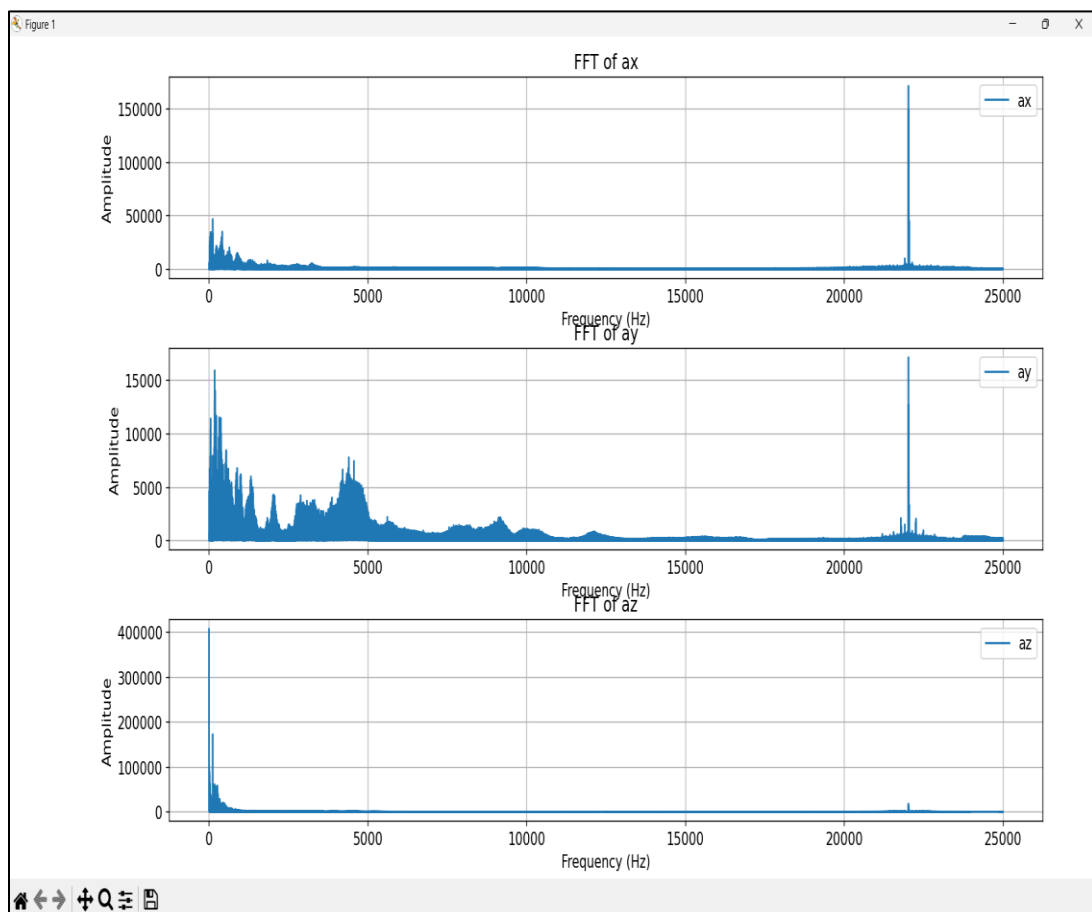


Figure 46.0: FFT for normal data

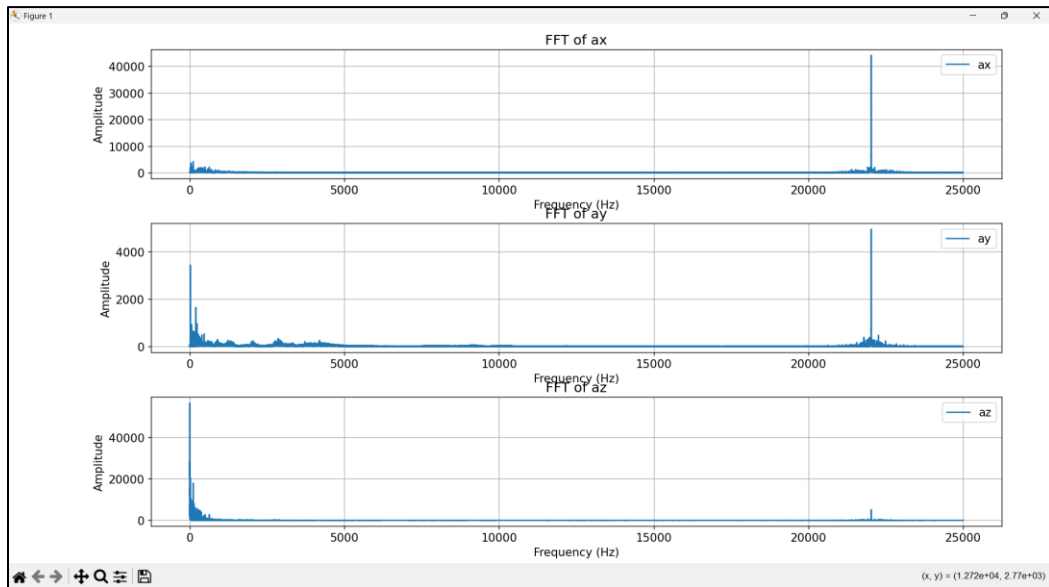


Figure 47.0: FFT for new data

### 5.5.2 Machine Learning Module

To determine the types of faults when they occur, a model was trained using vibration data to predict the kind of fault. The code to train the model performed an FFT and extracted features such as mean, standard deviation, and maximum amplitude values. The data was preprocessed, labeled, and split into training and testing sets. The ANN multi-class model was then built using TensorFlow's Keras and consisted of dense layers, batch normalization, and dropout layers. GridSearchCV tunes hyperparameters like batch size, epochs, and optimizer. The confusion matrix Figure 48.0 is displayed as percentages, showing the proportion of each predicted class compared to the actual class. The classification report Figure 49.0 showed precision, recall, F1-score, and support for each class. The classes considered were:

0 – Normal Operation

1 - Imbalance Fault

2 – Vertical Misalignment Fault

3 – Horizontal Misalignment Fault

4 – Underhang Fault

5 – Overhang Fault

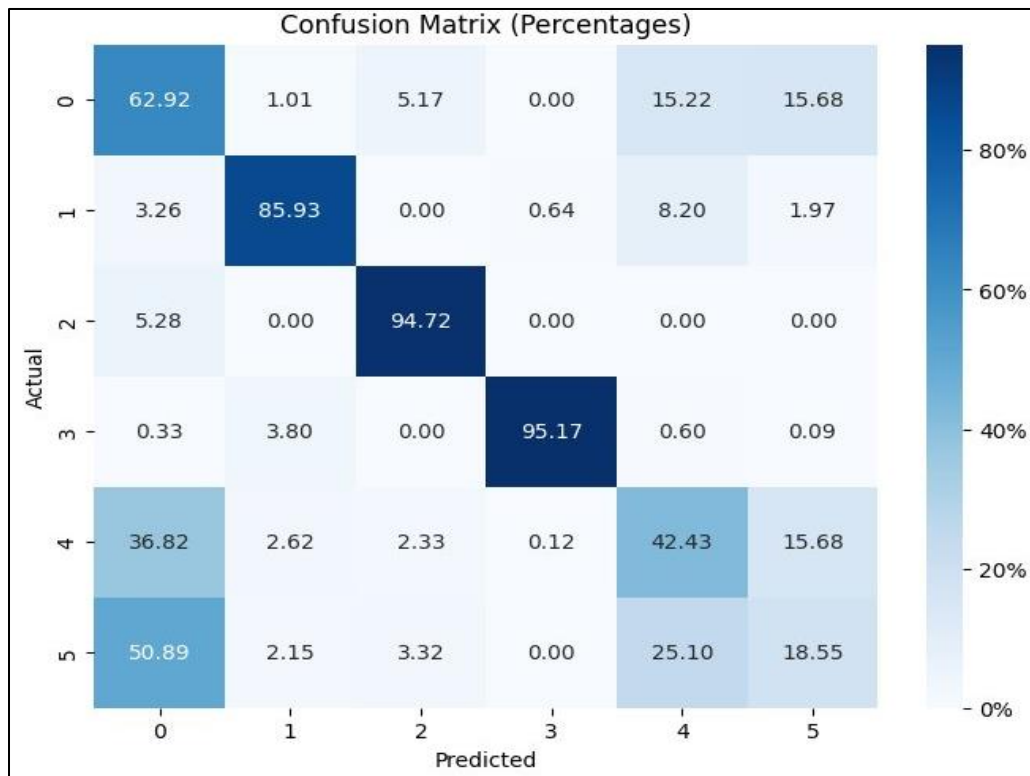


Figure 48.0: Confusion matrix

	precision	recall	f1-score	support
0	0.40	0.63	0.49	7607
1	0.90	0.86	0.88	7477
2	0.90	0.95	0.92	7468
3	0.99	0.95	0.97	7467
4	0.47	0.42	0.44	7594
5	0.35	0.19	0.24	7387
accuracy			0.67	45000
macro avg	0.67	0.67	0.66	45000
weighted avg	0.67	0.67	0.66	45000

Figure 49.0: Classification report

## **Chapter 6.0: Conclusion & Future Work**

This chapter talks about the achievements of this project, the limitations, and future works to improve upon this project

### **6.1 Conclusion**

The goals of this project were real-time fault detection and diagnosis, predictive maintenance, remote monitoring and management, and energy-efficient optimization. At the end of the project, the design system accomplished real-time fault detection and diagnosis by performing FFTs and using condition statements for motor temperature that determined the occurrence of faults. Then, the machine learning code was used to identify the type of faults that occurred. The results were then displayed on the Raspberry Pi system. For predictive maintenance, the developed machine learning code to predict when faults occurred allowed the system to notify users when faults would occur and hence allow maintenance to happen before that. The monitoring and data processing results were stored in the database and then displayed on the website, allowing remote access. Finally, using MQTT allowed energy-efficient data transmission, and the Arduino code made the Esp32 sleep at one-hour intervals with twenty minutes of operating Time. This allowed efficient use of energy.

### **6.2 Limitations**

A few challenges were faced in this project, and success was limited in some categories. The challenges faced were:

- Inability to obtain a working AMG8833 in the area. The AMG8833 would have provided temperature in a 64-pixel format, but due to the failure to obtain it, the FLIR LEPTON camera had to be used, creating images with large sizes with a field



of view that was more than needed. The images had to be sent to the database by a separate Python script.

- Inability to create faulty data from a physical motor. This issue affected the ability to train and test the machine-learning model. The data sources for processing provided parameters different from those used by the overall system. They had a few parameters, like vibration (Kaggle dataset) or current (Simulink model).

### **6.3 Future Works**

To improve the project further, a few further actions can be undertaken. These are:

- Obtaining an induction motor and introducing faults to obtain data for all the parameters for each type of fault. This will help raise the model accuracies and make it reliable for different motors.
- The AMG8833 can be obtained, and the system can be changed to increase its efficiency further. This is due to the lower data size of the thermal images.
- Parameters such as smoke or ultrasonic noise can be monitored to improve the model's accuracy.
- A deeper look into predictive maintenance to predict when maintenance is needed.
- A system that is built during the manufacturing stage of induction motors to install a system already integrated into the motor.

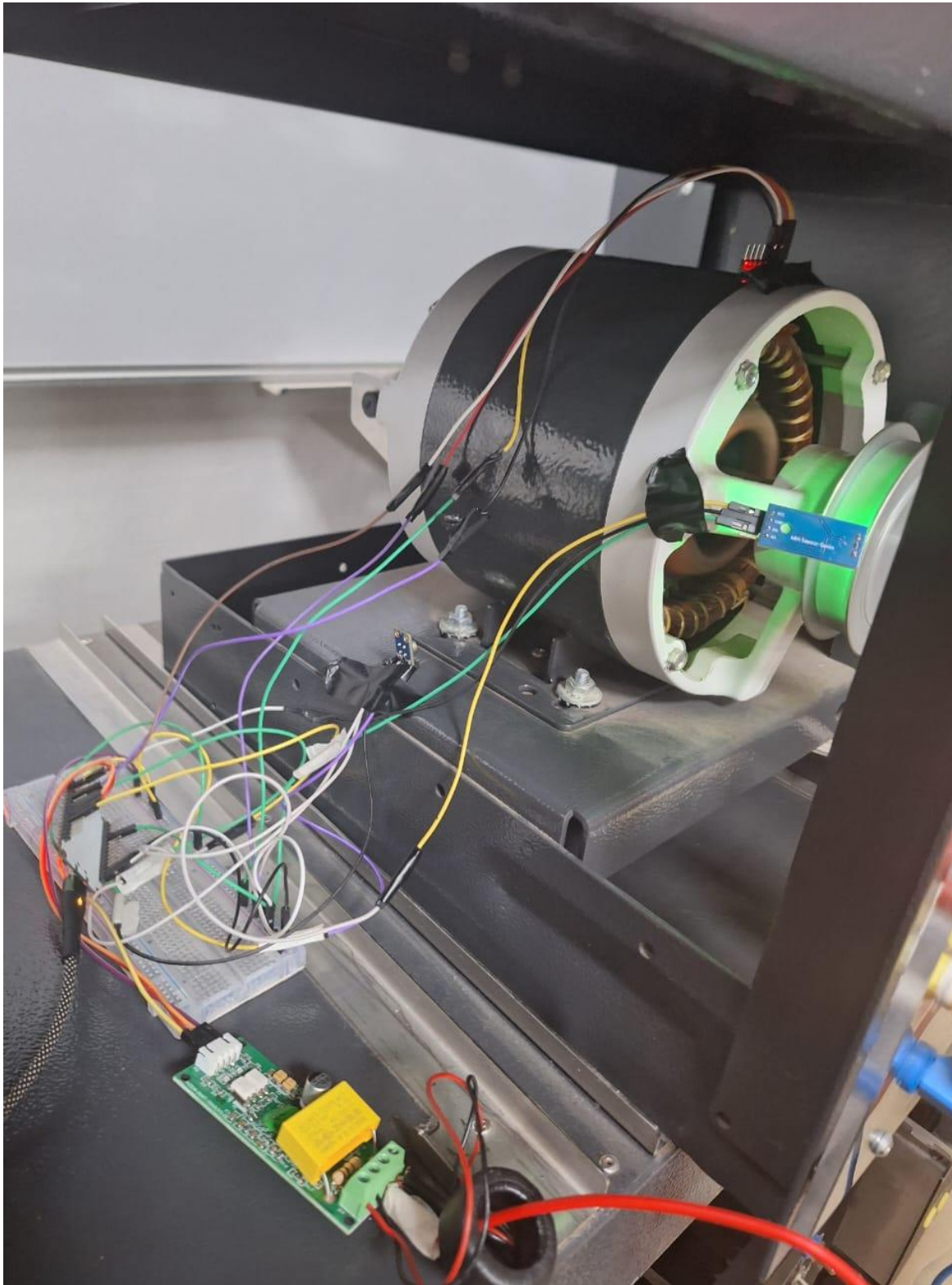
## References

- [1] D. Ganga and V. Ramachandran, "IoT-Based Vibration Analytics of Electrical Machines," *IEEE Internet Things J*, vol. 5, no. 6, pp. 4538–4549, Dec. 2018, doi: 10.1109/IIOT.2018.2835724.
- [2] H. A. Raja, H. Raval, T. Vaimann, A. Kallaste, A. Rassolkin, and A. Belahcen, "Cost-efficient real-time condition monitoring and fault diagnostics system for BLDC motor using IoT and Machine learning," in *2022 International Conference on Diagnostics in Electrical Engineering (Dignostika)*, IEEE, Sep. 2022, pp. 1–4. doi: 10.1109/Dignostika55131.2022.9905102.
- [3] D. Okeke and S. M. Musa, "Energy Management and Anomaly Detection in Condition Monitoring for Industrial Internet of Things Using Machine Learning," in *2021 International Conference on Informatics, Multimedia, Cyber and Information System (ICIMCIS)*, IEEE, Oct. 2021, pp. 65–68. doi: 10.1109/ICIMCIS53775.2021.9699352.
- [4] K. N. Gyftakis, D. V. Spyropoulos, J. C. Kappatou, and E. D. Mitronikas, "A Novel Approach for Broken Bar Fault Diagnosis in Induction Motors Through Torque Monitoring," *IEEE Transactions on Energy Conversion*, vol. 28, no. 2, pp. 267–277, Jun. 2013, doi: 10.1109/TEC.2013.2240683.
- [5] G. Enchev, N. Djararov, D. Tsvetanov, J. Djararova, and J. M. Guerrero, "Investigation of Machine Learning Classifiers for Diagnostics of Induction Motors," in *2023 IEEE International Conference on Environment and Electrical Engineering and 2023 IEEE Industrial and Commercial Power Systems Europe (EEEIC / I&CPS Europe)*, IEEE, Jun. 2023, pp. 1–7. doi: 10.1109/EEEIC/ICPSEurope57605.2023.10194685.
- [6] A. Almounajjed, A. K. Sahoo, M. K. Kumar, and M. W. Bakro, "Condition Monitoring and Fault Diagnosis of Induction Motor - An Experimental Analysis," in *2021 7th International Conference on Electrical Energy Systems (ICEES)*, IEEE, Feb. 2021, pp. 433–438. doi: 10.1109/ICEES51510.2021.9383729.
- [7] X. Liang, M. Z. Ali, and H. Zhang, "Fault Diagnosis for Induction Motors Using Finite Element Method - A Review," in *2019 IEEE Industry Applications Society Annual Meeting*, IEEE, Sep. 2019, pp. 1–11. doi: 10.1109/IAS.2019.8912449.
- [8] M. Juez-Gil, J. J. Saucedo-Dorantes, Á. Arnaiz-González, C. López-Nozal, C. García-Osorio, and D. Lowe, "Early and extremely early multi-label fault diagnosis in induction motors," *ISA Trans*, vol. 106, pp. 367–381, Nov. 2020, doi: 10.1016/j.isatra.2020.07.002.
- [9] W. Zhou, T. G. Habetler, and R. G. Harley, "Bearing Condition Monitoring Methods for Electric Machines: A General Review," in *2007 IEEE International Symposium on Diagnostics for Electric Machines, Power Electronics and Drives*, IEEE, Sep. 2007, pp. 3–6. doi: 10.1109/DEMPED.2007.4393062.
- [10] K. H. Hui, M. H. Lim, M. S. Leong, and S. M. Al-Obaidi, "Dempster-Shafer evidence theory for multi-bearing faults diagnosis," *Eng Appl Artif Intell*, vol. 57, pp. 160–170, Jan. 2017, doi: 10.1016/j.engappai.2016.10.017.

- [11] A. Glowacz and Z. Glowacz, "Diagnosis of the three-phase induction motor using thermal imaging," *Infrared Phys Technol*, vol. 81, pp. 7–16, Mar. 2017, doi: 10.1016/j.infrared.2016.12.003.
- [12] L. Magadán, F. J. Suárez, J. C. Granda, and D. F. García, "Low-Cost Industrial IoT System for Wireless Monitoring of Electric Motors Condition," *Mobile Networks and Applications*, vol. 28, no. 1, pp. 97–106, Feb. 2023, doi: 10.1007/s11036-022-02017-2.
- [13] H. Barksdale, Q. Smith, and M. Khan, "Condition Monitoring of Electrical Machines with Internet of Things," in *SoutheastCon 2018*, IEEE, Apr. 2018, pp. 1–4. doi: 10.1109/SECON.2018.8478989.
- [14] F. Civerchia, S. Bocchino, C. Salvadori, E. Rossi, L. Maggiani, and M. Petracca, "Industrial Internet of Things monitoring solution for advanced predictive maintenance applications," *J Ind Inf Integr*, vol. 7, pp. 4–12, Sep. 2017, doi: 10.1016/j.jii.2017.02.003.
- [15] M. Kazmi, M. Tabasum Shoaib, A. Aziz, H. Raza Khan, and S. Ahmed Qazi, "An Efficient IIoT-Based Smart Sensor Node for Predictive Maintenance of Induction Motors," *Computer Systems Science and Engineering*, vol. 47, no. 1, pp. 255–272, 2023, doi: 10.32604/csse.2023.038464.
- [16] X. Liang and K. Edomwandekhoe, "Condition monitoring techniques for induction motors," in *2017 IEEE Industry Applications Society Annual Meeting*, IEEE, Oct. 2017, pp. 1–10. doi: 10.1109/IAS.2017.8101860.
- [17] S. Guedidi, S. E. Zouzou, W. Laala, M. Sahraoui, and K. Yahia, "Broken bar fault diagnosis of induction motors using MCSA and neural network," in *8th IEEE Symposium on Diagnostics for Electrical Machines, Power Electronics & Drives*, IEEE, Sep. 2011, pp. 632–637. doi: 10.1109/DEMPED.2011.6063690.

## Appendix

### Appendix A: Image of the entire system



## Appendix B: Arduino Code

```
#include <Wire.h>

#include <Adafruit_MLX90614.h>

#include <ArduinoJson.h>

#include <PubSubClient.h>

#include <WiFiClientSecure.h>

#include <PZEM004Tv30.h>

#include <ArduinoOTA.h> // Include the OTA library

#include <esp_sleep.h> // Include the ESP sleep library


Adafruit_MLX90614 mlx = Adafruit_MLX90614(); // Initialize GY-906


PZEM004Tv30 pzem(Serial2, 16, 17); // Initialize with Serial2, RX pin 16, and TX pin 17


const char* ssid = "Galaxy";

const char* password = "ampomah1";

const char* mqtt_server = "192.168.247.231"; // replace with your broker URL

const int mqtt_port = 1883;


WiFiClient espClient;

PubSubClient client(espClient);

unsigned long lastMsg = 0;

unsigned long lastVibrationMsg = 0;

unsigned long lastSpeedMsg = 0;

unsigned long lastVoltageCurrentMsg = 0;
```

```

unsigned long lastTemperatureMsg = 0;

#define MSG_BUFFER_SIZE (50)

char msg[MSG_BUFFER_SIZE];

const char* vibration_topic = "motor_data/vibration_data";

const char* speed_topic = "motor_data/speed";

const char* voltage_topic = "motor_data/voltage";

const char* current_topic = "motor_data/current";

const char* temperature_topic = "motor_data/temperature";

const char* command2_topic = "AshesiIoTTopic/#";

const char* command1_topic = "command1";

// New Hall effect sensor pin definitions

const int hallSensorPin = 35; // Hall effect sensor digital output connected to GPIO 35

// Sampling intervals in milliseconds

const int vibrationSamplingInterval = 1; // 1 kHz for vibration

const int speedSamplingInterval = 2; // 500 Hz for speed

const int voltageCurrentSamplingInterval = 1000; // 1 second for voltage and current

const int temperatureSamplingInterval = 1000; // 1 second for temperature

volatile int pulseCount = 0; // Pulse counter for speed measurement

// Time in microseconds to sleep (1 hour = 3,600,000,000 microseconds)

const uint64_t SLEEP_DURATION = 3600000000ULL;

```

```

// Run time duration in milliseconds (20 minutes = 1,200,000 milliseconds)

const unsigned long RUN_DURATION = 1200000;


unsigned long startTime = 0;


// Interrupt Service Routine for Hall sensor

void IRAM_ATTR countPulse() {

    pulseCount++;

}


void setup_wifi() {

    delay(10);

    Serial.print("\nConnecting to ");

    Serial.println(ssid);


    WiFi.mode(WIFI_STA);

    WiFi.begin(ssid, password);

    Serial.println("");


    while (WiFi.status() != WL_CONNECTED) {

        delay(500);

        Serial.print(".");

    }

```

```

Serial.println("");

Serial.print("Connected to ");

Serial.println(ssid);

Serial.print("IP address: ");

Serial.println(WiFi.localIP());

randomSeed(micros());

}

void reconnect() {

while (!client.connected()) {

    Serial.print("Attempting MQTT connection...");

    String clientId = "ESP32Client-"; // Create a random client ID

    clientId += String(random(0xffff), HEX); //you could make this static

    if (client.connect(clientId.c_str())) {

        Serial.println("connected");

        client.subscribe(command2_topic); // subscribe to the topics here

    } else {

        Serial.print("failed, rc=");

        Serial.print(client.state());

        Serial.println(" try again in 5 seconds");

        delay(5000);

    }

}

}

}

```



```

void setup() {

  Serial.begin(115200);

  Serial.println("Setting up");

  setup_wifi();


  // Setup OTA

  ArduinoOTA.onStart([]() {

    String type;

    if (ArduinoOTA.getCommand() == U_FLASH)
      type = "sketch";
    else // U_SPIFFS
      type = "filesystem";

    // NOTE: if updating SPIFFS this would be the place to unmount SPIFFS using
    SPIFFS.end()

    Serial.println("Start updating " + type);

  });

  ArduinoOTA.onEnd([]() {

    Serial.println("\nEnd");

  });

  ArduinoOTA.onProgress([](unsigned int progress, unsigned int total) {

    Serial.printf("Progress: %u%%\r", (progress / (total / 100)));

  });

  ArduinoOTA.onError([](ota_error_t error) {

    Serial.printf("Error[%u]: ", error);

    if (error == OTA_AUTH_ERROR) Serial.println("Auth Failed");
  });
}

```

```

else if (error == OTA_BEGIN_ERROR) Serial.println("Begin Failed");

else if (error == OTA_CONNECT_ERROR) Serial.println("Connect Failed");

else if (error == OTA_RECEIVE_ERROR) Serial.println("Receive Failed");

else if (error == OTA_END_ERROR) Serial.println("End Failed");

});

ArduinoOTA.begin();


if (!mlx.begin()) {

    Serial.println("Failed to find MLX90614 chip");

    while (1) {

        delay(10);

    }

}


client.setServer(mqtt_server, mqtt_port);

client.setCallback(callback);


pinMode(hallSensorPin, INPUT_PULLUP); // Set hall sensor pin as input with internal
pullup

attachInterrupt(digitalPinToInterrupt(hallSensorPin), countPulse, FALLING); // Attach
interrupt to the hall sensor pin


// Record the start time

startTime = millis();

}

```

```

void loop() {

  ArduinoOTA.handle(); // Handle OTA updates

  if (!client.connected()) reconnect();

  client.loop();

  unsigned long now = millis();

  // Check if the run duration has elapsed

  if (now - startTime >= RUN_DURATION) {

    Serial.println("Entering deep sleep mode for 1 hour...");

    esp_sleep_enable_timer_wakeup(SLEEP_DURATION);

    esp_deep_sleep_start();

  }

  // Sample speed data at 500 Hz

  if (now - lastSpeedMsg >= speedSamplingInterval) {

    lastSpeedMsg = now;

    // Calculate RPM from pulse count

    int pulse = pulseCount;

    pulseCount = 0;

    float rpm = (pulse * 60.0) / (speedSamplingInterval / 1000.0); // Convert pulses to RPM

    // Create a JSON document for speed data

```

```

StaticJsonDocument<100> speedDoc;

speedDoc["RPM"] = rpm;


// Serialize JSON document to string
char speedBuffer[100];

serializeJson(speedDoc, speedBuffer);


// Publish speed data
publishMessage(speed_topic, String(speedBuffer), true);


// Print RPM to Serial for debugging
Serial.print("RPM: ");

Serial.println(rpm);
}


// Sample voltage and current data at 1 Hz
if (now - lastVoltageCurrentMsg >= voltageCurrentSamplingInterval) {

    lastVoltageCurrentMsg = now;


// Read voltage and current from PZEM-004T
float voltage = pzem.voltage();

float current = pzem.current();


// Create JSON documents for voltage and current data
StaticJsonDocument<100> voltageDoc;

```

```

voltageDoc["Voltage"] = voltage;

char voltageBuffer[100];

serializeJson(voltageDoc, voltageBuffer);

publishMessage(voltage_topic, String(voltageBuffer), true);

Serial.print("Voltage: ");

Serial.println(voltage);


StaticJsonDocument<100> currentDoc;

currentDoc["Current"] = current;

char currentBuffer[100];

serializeJson(currentDoc, currentBuffer);

publishMessage(current_topic, String(currentBuffer), true);

Serial.print("Current: ");

Serial.println(current);

}


// Sample temperature data at 1 Hz

if (now - lastTemperatureMsg >= temperatureSamplingInterval) {

    lastTemperatureMsg = now;


    // Get temperature from GY-906

    float temperatureC = mlx.readObjectTempC();


    // Create a JSON document for temperature data

    StaticJsonDocument<100> temperatureDoc;

```

```

    temperatureDoc["Temperature_C"] = temperatureC;

    char temperatureBuffer[100];

    serializeJson(temperatureDoc, temperatureBuffer);

    publishMessage(temperature_topic, String(temperatureBuffer), true);

    Serial.print("Temperature (C): ");

    Serial.println(temperatureC);

}

}

void callback(char* topic, byte* payload, unsigned int length) {

    String incommingMessage = "";

    for (int i = 0; i < length; i++) incommingMessage += (char)payload[i];

    Serial.println("Message arrived [" + String(topic) + "]" + incommingMessage);

    if (strcmp(topic, command1_topic) == 0) {

        if (incommingMessage.equals("1")) digitalWrite(BUILTIN_LED, LOW); // Turn the
LED on

        else digitalWrite(BUILTIN_LED, HIGH); // Turn the LED off

    }

}

void publishMessage(const char* topic, String payload, boolean retained) {

    if (client.publish(topic, payload.c_str(), true))

        Serial.println("Message published [" + String(topic) + "]: " + payload);}

```

## Appendix C: Code to subscribe and post mqtt data

```
import paho.mqtt.client as mqtt
import requests
import json

# MQTT Broker
mqtt_broker = "192.168.247.231"
mqtt_port = 1883
mqtt_main_topic = "motor_data"

# Define the IP address of the API server
api_ip = "192.168.247.198" # Replace this with the IP address of your API server

# Corrected API Endpoints (ensure these URLs point to your actual web server)
api_url_current = f"http://{api_ip}/Capstone/3wObjectsRest/current_read.php"
api_url_voltage = f"http://{api_ip}/Capstone/3wObjectsRest/voltage.php"
api_url_speed = f"http://{api_ip}/Capstone/3wObjectsRest/speed.php"
api_url_vibration = f"http://{api_ip}/Capstone/3wObjectsRest/vibration_api.php"
api_url_temperature = f"http://{api_ip}/Capstone/3wObjectsRest/temperature.php" # Add the temperature API endpoint

# MQTT on_connect callback
def on_connect(client, userdata, flags, rc):
    print(f"Connected with result code {rc}")
    client.subscribe(mqtt_main_topic + "/" + "#") # Subscribe to all subtopics of the main topic

# Function to send data to the appropriate API
def send_data(api_url, data):
    try:
        print(f"Sending data to URL: {api_url} with payload: {data}") # Debug information
        response = requests.post(api_url, json=data)
        print(f"API Response: {response.status_code}, Response Text: {response.text}")
    except Exception as e:
        print(f"Error sending data to API: {str(e)}")

# MQTT on_message callback
def on_message(client, userdata, msg):
```

```

# MQTT on_message callback
def on_message(client, userdata, msg):
    payload = msg.payload.decode()
    print(f"Received message: {payload} on topic {msg.topic}")

    # Extract subtopics
    subtopics = msg.topic.split("/")

    if len(subtopics) == 2: # Ensure the topic has the expected number of subtopics
        _, parameter_name = subtopics

        # Parse the payload as JSON
        try:
            data = json.loads(payload)
        except json.JSONDecodeError:
            print("Error decoding JSON")
            return

        # Determine the API endpoint and data structure
        if parameter_name == "current":
            api_url = api_url_current
            data_payload = {
                "C": data["Current"]
            }
        elif parameter_name == "voltage":
            api_url = api_url_voltage
            data_payload = {
                "V": data["Voltage"]
            }
        elif parameter_name == "speed_data":
            api_url = api_url_speed
            data_payload = {
                "S": data["RPM"]
            }
        elif parameter_name == "vibration_data":

```



```

elif parameter_name == "vibration_data":
    api_url = api_url_vibration
    data_payload = {
        "ax": data["AccelX"],
        "ay": data["AccelY"],
        "az": data["AccelZ"]
    }
elif parameter_name == "temperature":
    api_url = api_url_temperature
    data_payload = {
        "temp": data["Temperature_C"]
    }
else:
    print("Unknown parameter name")
    return

# Send data to the appropriate API
send_data(api_url, data_payload)
else:
    print("Unexpected topic structure")

# Create MQTT client instance
client = mqtt.Client()
client.on_connect = on_connect
client.on_message = on_message

# Connect to MQTT broker
client.connect(mqtt_broker, mqtt_port, 60)

# Loop to maintain connection
client.loop_forever()

```

## Appendix D: Machine Learning Code

```
import os
import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.metrics import classification_report, confusion_matrix,
accuracy_score
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, BatchNormalization
from sklearn.base import BaseEstimator, ClassifierMixin
import matplotlib.pyplot as plt
import seaborn as sns
from functools import partial

class KerasClassifier(BaseEstimator, ClassifierMixin):
    def __init__(self, build_fn=None, epochs=10, batch_size=32, **kwargs):
        self.build_fn = build_fn
        self.epochs = epochs
        self.batch_size = batch_size
        self.kwargs = kwargs
        self.model = None

    def fit(self, X, y, **kwargs):
        if self.model is None:
            self.model = self.build_fn(**self.kwargs)
            self.history = self.model.fit(X, y, epochs=self.epochs,
batch_size=self.batch_size, **kwargs)
        return self

    def predict(self, X, **kwargs):
        return np.argmax(self.model.predict(X, **kwargs), axis=-1)

    def predict_proba(self, X, **kwargs):
        return self.model.predict(X, **kwargs)

    def score(self, X, y, **kwargs):
        return accuracy_score(y, self.predict(X, **kwargs))

def load_data_from_directory(directory, label, features, file_limit=1):
    data_list = []
    print(f>Loading data from directory: {directory}")
    file_count = 0
    for root, _, files in os.walk(directory):
        for filename in files:
            if filename.endswith('.csv'):
                file_path = os.path.join(root, filename)
```

```

        print(f"Reading file: {file_path}")
        df = pd.read_csv(file_path, usecols=[1, 2, 3],
names=features)
        df = df.sample(frac=1).reset_index(drop=True)
        df['label'] = label
        data_list.append(df)
        file_count += 1
        if file_count >= file_limit:
            break
    if file_count >= file_limit:
        break
    if not data_list:
        print(f"No CSV files found in directory: {directory}")
    print(f"Loaded {len(data_list)} files from {directory}")
    return pd.concat(data_list, ignore_index=True) if data_list else
pd.DataFrame(columns=features + ['label'])

def perform_fft(vibration_data):
    fft_features = []
    for axis in ['ax', 'ay', 'az']:
        values = vibration_data[axis].values if
isinstance(vibration_data[axis], pd.Series) else [vibration_data[axis]]
        fft_vals = np.abs(np.fft.fft(values))
        fft_features.append(np.mean(fft_vals))
        fft_features.append(np.std(fft_vals))
        fft_features.append(np.max(fft_vals))
    return fft_features

# Define the directories for each condition
data_directories = {
    'normal': r'C:\Users\234561\OneDrive - Ashesi
University\Desktop\School\Capstone Docs\archive\normal\normal',
    'imbalance': r'C:\Users\234561\OneDrive - Ashesi
University\Desktop\School\Capstone Docs\archive\imbalance\imbalance'
}

# Define the features you are interested in
features = ['ax', 'ay', 'az']

# Load and combine the data
combined_data = pd.DataFrame()
for label, directory in data_directories.items():
    print(f"Loading data for label: {label}")
    combined_data = pd.concat([combined_data,
load_data_from_directory(directory, label, features)], ignore_index=True)

# Display the first few rows of the combined Dataset to understand its
structure

```

```

print("Data loading completed.")
print(combined_data.head())

# Perform FFT on vibration data and add as features
print("Performing FFT on vibration data...")
fft_features = combined_data.apply(lambda row: perform_fft(row), axis=1)
fft_features_df = pd.DataFrame(fft_features.tolist(),
columns=['fft_ax_mean', 'fft_ax_std', 'fft_ax_max', 'fft_ay_mean',
'fft_ay_std', 'fft_ay_max', 'fft_az_mean', 'fft_az_std', 'fft_az_max'])
combined_data = pd.concat([combined_data, fft_features_df], axis=1)
print("FFT completed.")
print(combined_data.head())

# Define final features
final_features = ['fft_ax_mean', 'fft_ax_std', 'fft_ax_max', 'fft_ay_mean',
'fft_ay_std', 'fft_ay_max', 'fft_az_mean', 'fft_az_std', 'fft_az_max']

# Splitting data into features and target for multi-class classification
X = combined_data[final_features]
y = combined_data['label']

# Encode the labels as integers
label_encoder = LabelEncoder()
y_encoded = label_encoder.fit_transform(y)

# Normalize the data
print("Normalizing the data...")
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Save the scaler parameters to a CSV file
print("Saving scaler parameters...")
scaler_params = pd.DataFrame({
    'mean': scaler.mean_,
    'scale': scaler.scale_
})
scaler_params.to_csv('scaler_params.csv', index=False)

# Split the data into training and testing sets
print("Splitting data into training and testing sets...")
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y_encoded,
test_size=0.2, random_state=42)

# Function to create model for multi-class classification
def create_multiclass_model(optimizer='adam'):
    model = Sequential()
    model.add(Dense(units=128, activation='relu',
input_shape=(X_train.shape[1],)))

```

```

        model.add(BatchNormalization())
        model.add(Dense(units=64, activation='relu'))
        model.add(BatchNormalization())
        model.add(Dense(units=32, activation='relu'))
        model.add(Dropout(0.5))
        model.add(Dense(units=len(np.unique(y_train)),
activation='softmax')) # Softmax for multi-class classification
        model.compile(optimizer=optimizer,
loss='sparse_categorical_crossentropy', metrics=['accuracy'])
        return model

# Create Keras model for hyperparameter tuning
print("Creating Keras model for hyperparameter tuning...")

# Partial functions for different optimizers
build_fn_adam = partial(create_multiclass_model, optimizer='adam')
build_fn_rmsprop = partial(create_multiclass_model, optimizer='rmsprop')

keras_model = KerasClassifier(build_fn=create_multiclass_model, verbose=0)

# Define the grid search parameters
param_grid = {
    'batch_size': [16],
    'epochs': [1],
    'build_fn': [build_fn_adam, build_fn_rmsprop] # Different build
functions
}

print("Starting Grid Search...")
grid = GridSearchCV(estimator=keras_model, param_grid=param_grid, n_jobs=-
1, cv=3)
grid_result = grid.fit(X_train, y_train)

# Summarize results of hyperparameter tuning
print(f"Best: {grid_result.best_score_} using {grid_result.best_params_}")

# Create the best model
best_optimizer = grid_result.best_params_['build_fn'].keywords['optimizer']
best_model = create_multiclass_model(optimizer=best_optimizer)
print("Training the best model...")
best_model.fit(X_train, y_train, epochs=grid_result.best_params_['epochs'],
batch_size=grid_result.best_params_['batch_size'], verbose=0)
print("Model training completed.")

# Save the best multi-class classification model
print("Saving the best model...")
best_model.save('multiclass_model.h5')

```

```

# Evaluate the model on test data
print("Evaluating the model on test data...")
loss, accuracy = best_model.evaluate(X_test, y_test)
print(f'Test Accuracy: {accuracy * 100:.2f}%')

# Make predictions
print("Making predictions on test data...")
y_pred = best_model.predict(X_test)
y_pred = np.argmax(y_pred, axis=1) # Get the index of the max probability

# Classification report
print("Generating classification report...")
print(classification_report(y_test, y_pred))

# Confusion matrix
print("Generating confusion matrix...")
cm = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:")
print(cm)

# Convert confusion matrix to percentages
cm_percentage = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis] * 100

# Plotting the confusion matrix as percentages
print("Plotting confusion matrix in percentages...")
plt.figure(figsize=(8, 6))
sns.heatmap(cm_percentage, annot=True, fmt='.2f', cmap='Blues',
cbar_kws={'format': '%.0f%%'})
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix (Percentages)')
plt.show()

```

## Appendix E: Machine Learning Classification Code

```
import os

import pandas as pd

import numpy as np

from sklearn.preprocessing import StandardScaler, LabelEncoder

from sklearn.model_selection import train_test_split, GridSearchCV

from sklearn.metrics import classification_report, confusion_matrix, accuracy_score

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Dropout, BatchNormalization

from sklearn.base import BaseEstimator, ClassifierMixin

import matplotlib.pyplot as plt

import seaborn as sns

from functools import partial


class KerasClassifier(BaseEstimator, ClassifierMixin):

    def _init_(self, build_fn=None, epochs=10, batch_size=32, **kwargs):

        self.build_fn = build_fn

        self.epochs = epochs

        self.batch_size = batch_size

        self.kwargs = kwargs

        self.model = None

    def fit(self, X, y, **kwargs):

        if self.model is None:
```

```

        self.model = self.build_fn(**self.kwargs)

        self.history = self.model.fit(X, y, epochs=self.epochs, batch_size=self.batch_size,
**kwargs)

        return self

def predict(self, X, **kwargs):

    return np.argmax(self.model.predict(X, **kwargs), axis=-1)

def predict_proba(self, X, **kwargs):

    return self.model.predict(X, **kwargs)

def score(self, X, y, **kwargs):

    return accuracy_score(y, self.predict(X, **kwargs))

def load_data_from_directory(directory, label, features, file_limit=1):

    data_list = []

    print(f"Loading data from directory: {directory}")

    file_count = 0

    for root, _, files in os.walk(directory):

        for filename in files:

            if filename.endswith('.csv'):

                file_path = os.path.join(root, filename)

                print(f"Reading file: {file_path}")

                df = pd.read_csv(file_path, usecols=[1, 2, 3], names=features)

                df = df.sample(frac=0.5).reset_index(drop=True)

```



```

        df['label'] = label

        data_list.append(df)

        file_count += 1

        if file_count >= file_limit:

            break

    if file_count >= file_limit:

        break

if not data_list:

    print(f"No CSV files found in directory: {directory}")

print(f"Loaded {len(data_list)} files from {directory}")

return pd.concat(data_list, ignore_index=True) if data_list else
pd.DataFrame(columns=features + ['label'])

def perform_fft(vibration_data):

    fft_features = []

    for axis in ['ax', 'ay', 'az']:

        values = vibration_data[axis].values if isinstance(vibration_data[axis], pd.Series) else
[vibration_data[axis]]

        fft_vals = np.abs(np.fft.fft(values))

        fft_features.append(np.mean(fft_vals))

        fft_features.append(np.std(fft_vals))

        fft_features.append(np.max(fft_vals))

    return fft_features

# Define the directories for each condition

```

```

data_directories = {

    'normal': r'C:\Users\234561\OneDrive - Ashesi University\Desktop\School\Capstone
Docs\archive\normal\normal',

    'imbalance': r'C:\Users\234561\OneDrive - Ashesi University\Desktop\School\Capstone
Docs\archive\imbalance\imbalance',

    'class3': r'C:\Users\234561\OneDrive - Ashesi University\Desktop\School\Capstone
Docs\archive\imbalance\overhang', # Add the correct paths for the new classes

    'class4': r'C:\Users\234561\OneDrive - Ashesi University\Desktop\School\Capstone
Docs\archive\imbalance\vertical-misalignment',

    'class5': r'C:\Users\234561\OneDrive - Ashesi University\Desktop\School\Capstone
Docs\archive\imbalance\horizontal-misalignment'

}

# Define the features you are interested in

features = ['ax', 'ay', 'az']

# Load and combine the data

combined_data = pd.DataFrame()

for label, directory in data_directories.items():

    print(f"Loading data for label: {label}")

    combined_data = pd.concat([combined_data, load_data_from_directory(directory, label,
features)], ignore_index=True)

# Display the first few rows of the combined dataset to understand its structure

print("Data loading completed.")

```

```

print(combined_data.head())

# Perform FFT on vibration data and add as features

print("Performing FFT on vibration data...")

fft_features = combined_data.apply(lambda row: perform_fft(row), axis=1)

fft_features_df = pd.DataFrame(fft_features.tolist(), columns=['fft_ax_mean', 'fft_ax_std',
'fft_ax_max', 'fft_ay_mean', 'fft_ay_std', 'fft_ay_max', 'fft_az_mean', 'fft_az_std',
'fft_az_max'])

combined_data = pd.concat([combined_data, fft_features_df], axis=1)

print("FFT completed.")

print(combined_data.head())

# Define final features

final_features = ['fft_ax_mean', 'fft_ax_std', 'fft_ax_max', 'fft_ay_mean', 'fft_ay_std',
'fft_ay_max', 'fft_az_mean', 'fft_az_std', 'fft_az_max']

# Splitting data into features and target for multi-class classification

X = combined_data[final_features]

y = combined_data['label']

# Encode the labels as integers

label_encoder = LabelEncoder()

y_encoded = label_encoder.fit_transform(y)

# Normalize the data

```

```

print("Normalizing the data...")

scaler = StandardScaler()

X_scaled = scaler.fit_transform(X)


# Save the scaler parameters to a CSV file

print("Saving scaler parameters...")

scaler_params = pd.DataFrame({

    'mean': scaler.mean_,

    'scale': scaler.scale_

})

scaler_params.to_csv('scaler_params.csv', index=False)


# Split the data into training and testing sets

print("Splitting data into training and testing sets...")

X_train, X_test, y_train, y_test = train_test_split(X_scaled, y_encoded, test_size=0.2,

random_state=42)


# Function to create model for multi-class classification

def create_multiclass_model(optimizer='adam'):

    model = Sequential()

    model.add(Dense(units=128, activation='relu', input_shape=(X_train.shape[1],)))

    model.add(BatchNormalization())

    model.add(Dense(units=64, activation='relu'))

    model.add(BatchNormalization())

    model.add(Dense(units=32, activation='relu'))

```

```

    model.add(Dropout(0.5))

    model.add(Dense(units=len(np.unique(y_train)), activation='softmax')) # Softmax for
multi-class classification

    model.compile(optimizer=optimizer, loss='sparse_categorical_crossentropy',
metrics=['accuracy'])

    return model

# Create Keras model for hyperparameter tuning
print("Creating Keras model for hyperparameter tuning...")

# Partial functions for different optimizers
build_fn_adam = partial(create_multiclass_model, optimizer='adam')
build_fn_rmsprop = partial(create_multiclass_model, optimizer='rmsprop')

keras_model = KerasClassifier(build_fn=create_multiclass_model, verbose=0)

# Define the grid search parameters
param_grid = {
    'batch_size': [16],
    'epochs': [1],
    'build_fn': [build_fn_adam, build_fn_rmsprop] # Different build functions
}

print("Starting Grid Search...")

grid = GridSearchCV(estimator=keras_model, param_grid=param_grid, n_jobs=-1, cv=3)

```

```

grid_result = grid.fit(X_train, y_train)

# Summarize results of hyperparameter tuning
print(f"Best: {grid_result.best_score_} using {grid_result.best_params_}")

# Create the best model
best_optimizer = grid_result.best_params_['build_fn'].keywords['optimizer']
best_model = create_multiclass_model(optimizer=best_optimizer)
print("Training the best model...")
best_model.fit(X_train, y_train, epochs=grid_result.best_params_['epochs'],
batch_size=grid_result.best_params_['batch_size'], verbose=0)
print("Model training completed.")

# Save the best multi-class classification model
print("Saving the best model...")
best_model.save('multiclass_model.h5')

# Evaluate the model on test data
print("Evaluating the model on test data...")
loss, accuracy = best_model.evaluate(X_test, y_test)
print(f"Test Accuracy: {accuracy * 100:.2f}%")

# Make predictions
print("Making predictions on test data...")
y_pred = best_model.predict(X_test)

```

```

y_pred = np.argmax(y_pred, axis=1) # Get the index of the max probability

# Classification report
print("Generating classification report...")
print(classification_report(y_test, y_pred))

# Confusion matrix
print("Generating confusion matrix...")
cm = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:")
print(cm)

# Convert confusion matrix to percentages
cm_percentage = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis] * 100

# Plotting the confusion matrix as percentages
print("Plotting confusion matrix in percentages...")
plt.figure(figsize=(8, 6))
sns.heatmap(cm_percentage, annot=True, fmt='.2f', cmap='Blues', cbar_kws={'format':
'%.0f%%'})
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix (Percentages)')
plt.show()

```