Prediction and Classification of IoT Sensor Faults using Hybrid Deep Learning Model

Adisu Mulu Seba

A Thesis Submitted to

The Department of Computer Science and Engineering

School of Electrical Engineering and Computing

Presented in Partial Fulfillment of the Requirement for the Degree of Master's in

Computer Science and Engineering

Office of Graduate Studies

Adama Science and Technology University

June, 2023

Adama, Ethiopia

Prediction and Classification of IoT Sensor Faults using Hybrid Deep Learning Model

Adisu Mulu Seba

Advisor: Ketema Adere (PhD)

A Thesis Submitted to

The Department of Computer Science and Engineering

School of Electrical Engineering and Computing

Presented in Partial Fulfillment of the Requirement for the Degree of Master's in

Computer Science and Engineering

Office of Graduate Studies

Adama Science and Technology University

June, 2023

Adama, Ethiopia

Approval Page of M.Sc. Thesis

I, the advisor of the thesis entitled "**Prediction and Classification of IoT Sensor Faults using Hybrid Deep Learning Model"** and developed by Adisu Mulu, hereby certify that the recommendation and suggestions made by the board of examiners are appropriately incorporated into the final version of the thesis.

Dr. Ketema Adere	_____	_____
Major Advisor	Signature	Date

We, the undersigned, members of the Board of Examiners of the thesis by Adisu Mulu have read and evaluated the thesis entitled "**Prediction and Classification of IoT Sensor Faults using Hybrid Deep Learning Model"** and examined the candidate during the open defense. This is, therefore, to certify that the thesis is accepted for the partial fulfillment of the requirement of the degree of Master of Science in Computer Science and Engineering (CSE).

_____	_____	_____
Chairperson	Signature	Date
_____	_____	_____
Internal Examiner	Signature	Date
_____	_____	_____
External Examiner	Signature	Date

Finally, approval and acceptance of the thesis are contingent upon the submission of its final copy to the Office of Postgraduate Studies (OPGS) through the Department Graduate Council (DGC) and School Graduate Committee (SGC).

_____	_____	_____
Department Head	Signature	Date
_____	_____	_____
School Dean	Signature	Date
_____	_____	_____
Office of Postgraduate Studies, Dean	Signature	Date

Declaration

I hereby declare that this Master Thesis entitled "**Prediction and Classification of IoT Sensor Faults using Hybrid Deep Learning Model"** is my original work. That is, it has not been submitted for the award of any academic degree, diploma, or certificate in any other university. All sources of materials that are used for this thesis have been duly acknowledged through citation.

Adisu Mulu		
Name of the Student	Signature	Date

Recommendation

I, the advisor of this thesis, hereby certify that I have read the revised version of the thesis entitled "**Prediction and Classification of IoT Sensor Faults using Hybrid Deep Learning Model"** prepared under my guidance by Adisu Mulu submitted in partial fulfillment of the requirements for the degree of Masters of Science in Computer Science and Engineering (CSE). Therefore, I recommend the submission of the revised version of the thesis to the department following the applicable procedures.

Ketema Adere (PhD)_____	_____	_____
Major Advisor	Signature	Date

ACKNOWLEDGEMENTS

First of all, I would like to thank our ever-loving father, for blessing me with health and long life to complete this work.

I would also like to express my gratitude to Dr Ketema Adere, my advisor, for his patience, guidance and counseling, from the very beginning to this day, especially, his motivation to make us start reading on research areas even during the summer, before the academic year begun.

I would also like to express my gratitude to ASTU Computer Science and Engineering staff, particularly, Dr Mesfin Abebe, for his keen follow up on our progress and motivating us to work hard and finish on time and Dilla University for sponsoring this opportunity.

I would also like to express my gratitude to my parents and siblings, for their unconditional support and prayers, and, my friends and colleagues who have motivated me through the months.

Last but not least, I would also like to express my deepest gratitude to the various computer science, data science, machine learning and deep learning online communities, particularly on Quora.com. They were patient in understanding my questions, providing me with answers and explanations and different resources that have helped me better understand and work on my research.

Table of Contents

LIST OF TABLES

LIST OF FIGURES

LIST OF EQUATIONS

LIST OF ACRONYMS and ABBREVIATIONS

1

1D CNN: 1 Dimensional Convolutional Neural Network

A

API: Application Programming Interface

AUC: Area Under Curve

ARIMA: Autoregressive Integrated Moving Average

B

BiLSTM: Bidirectional Long Short Term Memory

C

CDHMM: Continuous Density Hidden Markov model

CGM: Continuous Glucose Monitoring

CAE: Convolutional Autoencoder

CNN: Convolutional Neural Network

CFS: *Correlation-based Feature Selection*

D

DL: Deep Learning

DL-BiLSTM: Double Layer Bidirectional Long-Short Term Memory

G

GRU: Gated Recurrent Unit

GPU: Graphical Processing Unit

I

IoT: Internet of Things

K

KPCA: Kernel Principal Component Analysis

L

LDTI: linear discrete time invariant

LSTM: Long Short-Term Memory

M

ML: Machine Learning

MAPE: Mean Abs olute Percentage Error

MAE: Mean Absolute Error

MSE: Mean Square Error

MLP: Multi-Layer Perceptron

N

NLP: Natural Language Processing

R

RFID: Radio Frequency Identification

RNNs: *Recurrent Neural Networks*

RMSE: Root Mean Squared Error

S

SVM: Support Vector Machine

SMOTE: Synthetic Minority Over-Sampling Technique

T

TPU: Tensor Processing Unit

V

VAR: Vector Autoregression

W

WSN: Wireless Sensor Network

Abstract

The quality and reliability of IoT ecosystems heavily rely on accurate and dependable sensor data. However, sensors can produce erroneous and faulty measurements due to various factors like environmental disturbances, electrical noise and others. These can have significant consequences across different domains, including a threat to safety in critical systems. Though many researches have been conducted, the existing literature primarily focuses on fault detection in the sensor data, while fault detection is useful, it is a reactive approach that identifies faults after they have occurred, meaning that actions are taken after the fault has already impacted the system, potentially leading to negative consequences. In this study, we take a proactive approach by developing a multi-stage solution using hybrid deep learning models. In the first stage, we trained a CNN-LSTM model to forecast multi-step future sensor measurements based on historical data. In the second stage, we trained a CNN-MLP model using fault-injected sensor data to learn patterns associated with different fault types and classify new measurements accordingly. By passing the forecasted sensor values as input to the classification model and categorizing them as normal, bias, drift, random or poly-drift, we anticipate potential faults before they manifest. The raw dataset used is the publicly available Intel Lab data, which has been annotated and fault-injected by De Brun, El. To identify the most effective models, we experimented with different models. For regression, we evaluated GRU, LSTM, BiLSTM, CNN-GRU, CNN-LSTM, and CNN-BiLSTM, comparing their performance using root mean squared error (RMSE), mean squared error (MSE) and mean absolute error (MAE) with 2-split timeseries cross-validation. CNN-LSTM outperformed the other models with a MAE of 2.0957 for a 45 timesteps forecast. For the classification task, we evaluated CNN, MLP, and CNN-MLP using the metrics accuracy, precision, recall, and F1-score with 5 and 10-fold cross-validations. CNN-MLP outperformed the others with accuracy of 96.11% for bias, 99.33% for drift, 98.61% for random and 98.81% for poly-drift. The average accuracy across the 4 faults is 98.21%, which is a 0.3% increase from the baseline work 97.91%. By adopting a proactive approach to sensor fault prediction and classification, our research aims to enhance the reliability and efficiency of IoT systems, allowing for preventive measures to be taken before faults have a detrimental impact.

Keywords: IoT, Sensor faults, multi-step forecasting, Intel Lab data

CHAPTER ONE

INTRODUCTION

1.1. Background of the Study

The Internet of Things (IoT) has emerged as a game-changing technology in recent years, with its ability to connect devices and sensors to the internet, enabling them to communicate and interact with each other. Sensors are the primary driving force behind the IoT revolution. These tiny devices collect data on everything from temperature to humidity to motion and light, providing real-time insights that can be used to improve efficiency, automation and productivity. For example, in manufacturing, IoT-enabled sensors are used to monitor equipment performance, reduce downtime and improving overall equipment effectiveness (Yang et al., 2018).

In agriculture, IoT-enabled sensors and devices are helping farmers improve crop yield and quality, reduce water usage and prevent crop diseases (Xu et al., 2022). For example, The Yield[1], an Australian based agricultural technology company is using IoT sensors to monitor the moisture levels in soil and provide real-time insights to farmers. This enables farmers to optimize irrigation and reduce water wastage, leading to improved crop yields. The health industry without doubt is one of the most significant impacts of these sensor technologies, providing real time remote monitoring of health parameters and management of chronic diseases by using IoT-enabled fitness trackers and smartwatches and IoT-enabled medical devices such as pacemakers and insulin pumps. In transportation sensors in the vehicles are used to collect data on vehicle performance, driver behavior and road conditions among other things to enable smarter and safer driving (Muthuramalingam et al., 2019).

In smart cities, the application of sensors is endless from monitoring air and noise pollution to traffic monitoring to parking and smart buildings. For example, Park & Breathe, a system developed by France based company Flowbird[2], measures both air and noise pollution in urban areas. PlacePod is one example of a smart-parking-sensor system developed by PNI

[1] https://www.theyield.com/
[2] https://www.flowbird.group/smartcity/en/solutions/

Sensor Corp[3]. The system communicates real time parking data enabling accurate vehicle detection in parking spaces.

As IoT technology continues to evolve, we can expect to see even more transformation and impacts on these different industries in the years to come. However, one of the major challenges in managing IoT systems is the failure of the sensors to work correctly due to different faults. Sensors are the building block of the IoT world. They are the major component of the IoT infrastructure. These sensors can however, be exposed to different faults, threats and even risks of failures due to a variety of reasons, including physical damage, environmental factors, or wear and tear over time. The correct operation of the sensors plays a vital role in the overall performance of the system.

When a sensor fails to work correctly it produces corrupted data or erroneous reading or conflicting information. When the IoT system processes this corrupted sensor data, the overall performance of the system is compromised, making it inaccurate and unreliable sometimes even leading to total failure (Li et al., 2020). This can have significant impact. For example, in health care system, if a sensor that monitors a patient's heart rate or blood glucose level fails, it can result in improper medication dosing, leading to adverse health effects. In agriculture, the total failure or incorrect working of a moisture sensor can result in over-or-under-watering of crops, affecting crop growth and yield. In the energy sector for example, an incorrect working or failure of a temperature sensor in a power plant can cause overheating and equipment damage. In logistic for example, an incorrect working or failure of a temperature-controlled shipping container can lead to spoilage of perishable goods. In aerospace, a failure or faulty working of sensors responsible for monitoring and controlling critical systems such as flight control, navigation, and engine can lead to catastrophic consequences (Welcer et al., 2022). For example, in smart cities, a failure or incorrect working of a sensor in a traffic management system such as traffic light can lead to traffic accidents and congestion (Ali et al., 2021). In oil and gas pipelines, a failure or incorrect working of sensors that monitor the pressure and temperature of the pipelines could cause a leakage which can be an environmental disaster.

[3] https://www.pnicorp.com

In summary, sensor failures or incorrect workings can have severe consequences in different industries, ranging from damage to property, system downtime, increased maintenance cost to loss of revenue to life threatening situations. Which is why sensor fault prediction is such an important area of research.

The existing studies have suggested different approaches such as model-based, signature based and data-driven approaches for detecting sensor faults and potential failures. The model-based approach uses a mathematical model (set of equations and rules) of the system to estimate the true value of a sensor reading and detect sensor faults by comparing the predicted and measured values(Gao et al., 2015)(Habibi et al., 2019). The signature-based approach compares the current sensor reading with a pre-defined threshold value. If the reading exceeds the threshold, it's an indicative of a fault(Clouqueur et al., 2001). The data-driven approach analyzes the statistical properties of the sensor data to learn the underlying patterns and relationships between the sensor reading and the occurrence of faults (Subbaraj & Kannapiran, 2010). However, the existing research mainly focuses on sensor fault detection as opposed to prediction. The problem with fault detection is that it relies on identifying a problem after it has already occurred or when the sensor is no longer operating as it should. In addition, the studies lack generalizability as most of the models are trained for specific faults. Also, the studies that aim to predict multiple faults still require improvement in their accuracy.

To fill this gap, in this study we built a two-stage deep learning architecture. In the first stage a regression model is built using a hybrid deep learning CNN-LSTM model with attention mechanism to forecast future values of the sensors. The model is trained on the historical data of the sensors. In the second stage, a classification model is trained to classify the sensors data as normal or one of the 4 fault types. The forecasted values from the regression model are then passed to the classification model to determine the status of the values thereby anticipate potential faults.

1.2. Motivation of the Study

The Internet of Things has transformed the way we live our lives and the way businesses operate. It has brought about significant changes across different industries, revolutionizing the way we work, communicate, and interact with technology. As IoT technology continues to

evolve, we can expect even more significant changes to come to different industries, enabling us to live more connected and efficient lives.

However, as discussed above sensors are the major component of the IoT infrastructure. They are basically the building block of IoT systems, and the correct operation of these sensors are absolutely vital to the whole system. We have discussed some of the consequences that results from failed IoT systems.

The motivation for this study arises from the desire to capitalize on the benefits of IoT technology and to harness the potential of these technology to an even greater extent than we are currently using, while at the same time trying to minimize the impact that results when the system malfunctions or fails.

1.3. Statement of the Problem

Sensor faults has received considerable attention in the research community. The proper operation of the sensors is crucial to the system's overall functionality. Corrupted data compromises the performance of the overall system, making it unreliable and may even lead to total system failure. Depending on the application domain, the impact of such failure could result to damage of property to loss of revenue to, in worst case even loss of life. Many researches have been conducted to address the issue. For example,(Huang et al., 2021), using time frequency analysis and deep learning detected and classified 3 types of faults.(Uppal et al., 2021), by analyzing the existing data (historical data) using machine learning techniques performed binary classification of faults i.e., healthy or unhealthy. (Jana et al., 2022), by using Convolutional Neural Network (CNN) and Convolutional Autoencoder (CAE) performed detection and classification of 4 types of faults.

However, the existing literature mainly focuses on fault detection rather than prediction. While detecting a fault is useful it only addresses the problem after it has occurred, meaning that actions are taken after the fault has already impacted the system, potentially leading to negative consequences, for example, the Deepwater Horizon oil spill in 2010 and the Boeing 737 max crash in 2019. Additionally, the literature is limited in the number of faults considered. Many studies focus on specific faults and so they cannot be generalized to other faults. This is a problem if we consider the deployment or application of these systems in the real world because a real-world setting will likely exhibit multiple fault types at the same time.

Furthermore, the existing studies that aim to predict multiple faults still require improvement in their accuracy.

By adopting a 2-step proactive approach that anticipates potential faults before they manifest using hybrid deep models and attention mechanisms to enhance the predictive power of the models, our research aims to enhance the reliability and efficiency of IoT systems, allowing for preventive measures to be taken before faults have a detrimental impact.

1.4. Research Questions

In this study, we have addressed the following research questions.

RQ1: What are high performing deep learning algorithms for univariate multi-step timeseries forecasting, based on historical data according to evaluation metrics.

RQ2: How can a hybrid deep learning model be developed to predict and classify sensor data faults.

RQ3: How can feature engineering techniques be applied to improve the performance of deep learning models in the case of time-series data.

1.5. Objectives of the Study

1.5.1. General objective

The general objective of this research is to develop a model for predicting and classifying IoT sensor faults using a hybrid deep learning model.

1.5.2. Specific objectives

The specific objectives of this research that are performed in order to achieve the general objective are the following:

- Develop different deep learning models for univariate time-series forecasting.
- Identify the model that gives the highest performance for timeseries forecasting based on evaluation metrics for regression such RMSE, MSE and others.
- Develop different deep learning models for multi-class classification for sensor data fault.
- Identify the model that gives the highest performance for classifying the sensor data using classification metrics such as accuracy, precision, recall and f1.

- Incorporate rolling window and time-based features to the model and evaluate the impact of this feature engineering on the performance of the model.

1.6. Scope and Limitation

1.6.1. Scope

The scope of this study includes forecasting the future values of the sensors, then classifying the forecasted values as either normal or as one of the 4 fault types covered. The study deals with data centric faults. The model is trained with data from 10 sensors' measurement from Intel Lab. The type of sensors considered in this study is temperature sensors. The reason for choosing temperature sensors for this study is due to the availability of annotated, labeled benchmark dataset for temperature sensors. In the future when we find annotated, labeled benchmark dataset for other types of sensors, since the underlying principle of the model does not need to change, by using transfer learning, we can train the model of the data of these sensors and allow the model to work for multiple types of sensors.

1.6.2. Limitation

The study will only use the sensors historical data to forecast the future values of the sensors. So, this will be a univariate forecasting. The proposed system will only forecast 45 timesteps ahead, i.e., 1:30 hrs.

1.7. Significance of the Study

This study is important in all major industries where IoT is used, particularly where consistent temperature monitoring and control is highly required as the models are trained on temperature sensors. By identifying potential sensor faults before they occur, proactive maintenance and repair can be carried out, reducing downtime and costs, and generally improve the reliability and performance of the system.

1.8. Organization of the Thesis

The rest of the thesis is organized in the following ways

Chapter Two: In this chapter, we will discuss key points related to sensors and faults and then provide a comprehensive review of the different approaches taken to address the issue of sensor failure predictions by other researchers. Critically evaluate and examine related works

of previous study, their contribution, limitation and gaps in knowledge, as well works that further the understanding of the topic.

Chapter Three: In this chapter, we will discuss the method and technique applied in this research to design the intended solution. The methodology used such as data source, data analysis, and processing procedures are detailed in depth. The evaluation metrics used are also discussed.

Chapter Four: In this chapter, we will provide the overall design of the model, the proposed architecture.

Chapter Five: In this chapter we will provide the experimentation and implementation of the proposed solution. This will include the working implementation environment, model construction and implementation of performance evaluation.

Chapter Six: In this chapter we will discuss the results. Explain the outcome of the proposed solution. We will summarize the overall research finding, draw conclusions and make recommendations for future work. Discuss the contribution made and the application of the study.

CHAPTER TWO

LITERATURE REVIEW

2.1. IoT Sensors – An overview

Sensor technology plays a crucial role in the context of Internet of Things (IoT). The IoT refers to the network of physical devices, vehicles, home appliances, and other items that are embedded with sensors and other connectivity, which enable them to communicate and collect and exchange data(Sethi & Sarangi, 2017). Sensors are critical components of IoT infrastructure as they allow them to collect real-time data from the environment and convert it into digital signals that can be transmitted over the internet.

IoT sensors can be categorized in a number of ways.

- Method of measurement: According to the different mechanisms of measurement, IoT sensors can be categorized into electrical sensors, optical sensors, chemicals sensors, and gyroscopic sensors.
- Based on what they measure (Physical quantity measured): sound, light, temperature, electromagnetic radiation, acceleration, velocity, and color. These are a few examples of IoT sensor types based on what they measure.
- Based on whether or not they need an external power source (Energy usage): Sensors that need an external power source are classed as active, while those that don't need are classed as passive sensors. RFID readers are examples of passive sensors.
- Based on the type of output they produce: can be categorized as digital or analog sensors. Analog sensors have a continuously variable output while digital sensors have a discrete output.
- In terms of range of measurement (Accuracy): IoT sensors can be classified as high- and low- precision sensors. High precision measure very small changes and are typically used in controlled environments. Low precision which can only measure large changes are often used in harsh environments.

IoT sensors are designed to be connected to the internet and provide data to cloud-based applications. They typically use wireless communication protocols such as Wi-Fi, Bluetooth, Zigbee or Z-Wave to transmit data to the cloud, where it can be analyzed and acted

upon(Zaidan et al., 2018). This is what differentiates IoT sensors from normal or stand-alone sensors that are usually not connected to the internet, and typically only provide a local output signal, such as analog or digital signal. IoT sensors typically have 4 components: A sensing element that detects the physical quantity being measured. A transducer that converts the signal from the sensing element into an electrical signal that can be measured and processed. A wireless communication interface that enables the sensor to transmit data wirelessly to the cloud or another connected device and finally a power source(Xia, 2009).

IoT sensors are being used in a wide range of applications, including home automation, smart cities, manufacturing and industrial IoT among others(Javaid et al., 2021). In smart homes, temperature sensors, motion sensors, and security sensors are used to monitor and control temperature, lighting, security and other aspects of the home environment. For example, motion sensor can turn on lights when someone enters the room. In healthcare, IoT sensors are used for remote patient monitoring, tracking medication adherence, and monitoring vital signs. In transportation, IoT sensors are used for tracking vehicles, optimizing routes, and improving safety.

The architecture of an IoT sensor can vary depending on the specific design but generally, an IoT sensor architecture with in the frame of IoT includes the following components;

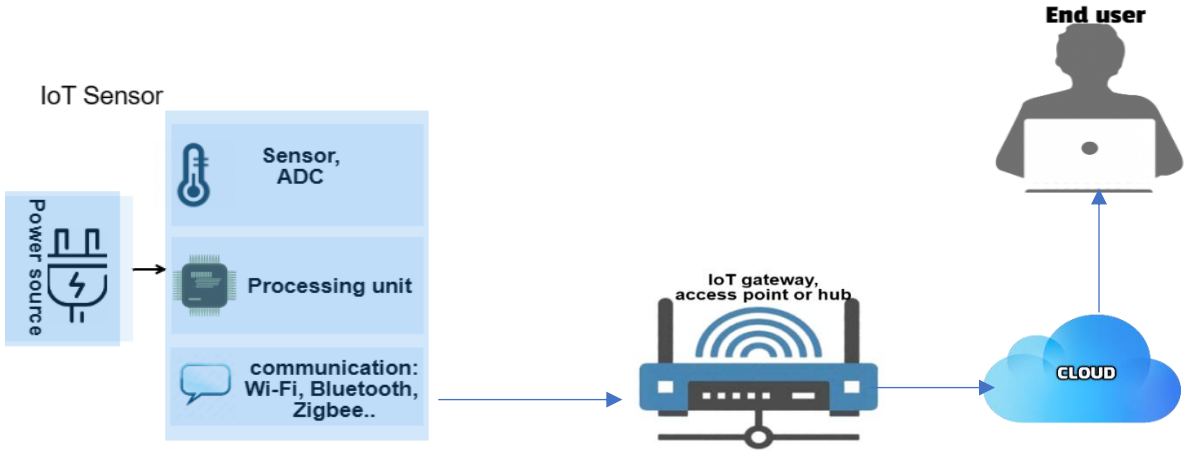


Figure 2-1.Typical IoT Sensor Architecture with in the frame of IoT

2.2. Sensor Faults

Sensor faults refers to a situation where a sensor deviates from its intended or expected behavior, leading to inaccurate or unreliable measurements and in some cases causing total

failures (Li et al., 2020). Sensor faults can occur due to a variety of reasons, including manufacturing defects, wear and tear, environmental factors, and software or firmware errors. For example, in manufacturing sensor faults can occur due to defects in the manufacturing process. These defects can lead to inaccurate sensor readings, or sensors that fail prematurely. In wear and tear sensors can deteriorate over time, which again can lead to inaccuracies or outright failures. This is particularly true for sensors that are exposed to harsh environments, such as extreme temperatures, moisture, or corrosive substances. Environment factors such as electromagnetic interference among others can also cause sensor faults by interfering with the sensor's signal or calibration. And errors in the software or firmware that controls the sensor can cause the sensor to operate incorrectly or fail altogether.

There are several approaches to categorizing faults, based on the severity of the fault, such as critical, major or minor faults. Based on the time of occurrence, such as permanents faults or transient. Permanent faults persist over time while transient faults are temporary. Based on location of the faults, such as internal or external faults. Based on the impact on the system such as, safety critical faults, operational faults, or maintenance faults. Two major approaches to classifying the faults are from the system point of view and the data point of view.

2.2.1. Data-centric view

This view focuses on the sensor data itself and classifies faults based on how they affect the data. The data-centric view considers each sensor output as an independent data stream and looks for anomalies in the data(Ni et al., 2009). This view is useful for detecting faults that affect the accuracy of the sensor data. A few examples include:

i. *Bias:* A bias is a consistent error in the sensor's output, where the sensor produces readings that are higher or lower than the actual value of the measured parameter. For example, a temperature sensor might consistently report temperatures that are 2 degrees Celsius higher than the actual temperature.
ii. *Drift:* A drift is a gradual shift in the sensor's output over time, where the sensor produces readings that are increasingly inaccurate. For example, a pH sensor might produce readings that become progressively more acidic over time.
iii. *Poly-drift:* A poly-drift is a combination of drift and bias, where the sensor's output shifts over time while maintaining a consistent offset from the true value. For example,

a barometer might consistently report pressure readings that are 5 hPa too high and drifts an additional 2 hPa per day.

iv. *Random:* Random errors are unpredictable fluctuations in the sensor's output that are not related to the measured parameter. For example, a digital scale might produce random fluctuations in its readings due to electrical noise in the sensor's circuitry.

2.2.2. System view

This view takes a broader perspective and considers the system as a whole, including all the components that interact with the sensor. The system view considers faults that can arise from the physical sensors themselves and their interaction between different components in the system. This view is useful for detecting faults that can affect the reliability or availability of the system (Ni et al., 2009). Examples include:

i. *Communication errors:* Communication errors can occur when sensors are unable to send or receive data correctly due to issues such as signal interference, incorrect wiring, or faulty communication protocols.

ii. *Power supply issues:* Power supply issues can affect the performance of sensors and sensor systems. For example, a low or unstable power supply can cause sensors to produce inaccurate readings or even fail completely.

iii. *Calibration issues:* Sensors may be calibrated incorrectly, leading to inaccurate readings. For example, a pressure sensor in a gas pipeline may produce incorrect measurements of gas pressure due to incorrectly being calibrated.

iv. *Sensor placement:* The poor placement of sensors can have significant impact. For example, installing a temperature sensor direct sunlight can result in inaccurate readings.

2.3. *Sensor Fault Prediction*

One of the key challenges in the context of IoT is detecting and diagnosing sensor faults in real time. This is because IoT systems typically involve large amounts of data that must be processed quickly and accurately to inform decision-making. In addition, IoT systems often operate in dynamic and unpredictable environments, which can make it difficult to distinguish between normal variations in sensor data and actual sensor faults. To address these challenges, researchers and engineers have studied different approaches to detecting and diagnosing

sensor faults. These approaches each have their own strength and weakness, and the specific conditions one would be preferred over the other. In addition to detecting and diagnosing sensor faults, researchers and engineers have also developed different approaches for mitigating the impact of sensor faults on data accuracy and reliability, such as the use of redundancy and fault-tolerant systems, which involve deploying multiple sensors to measure the same physical quantity and comparing readings to identify discrepancies.

2.3.1. *Model-based approach*

This approach uses a mathematical model of the system and compare the measured sensor data with the predicted values from the model (Gao et al., 2015)(Habibi et al., 2019). If there is a significant difference between the predicted and measured values, it is indicative of a sensor fault. This approach requires a priori knowledge of the system and its behavior to construct the mathematical model, and are often more accurate when the model is well defined. The model typically consists of set of equations or rules that capture the dynamics of the system and its interactions with the environment. (Hashimoto et al., 2003) propose a multi-modal approach to fault detection and diagnosis of internal sensor for mobile robot. Three failure modes (hard, noise and scale failure) of the sensor are handled. The mode probabilities, which are estimated based on a bank of Kalman filters, provide the fault decision which is made by comparing the model conditional estimates in the sensor gain. (He et al., 2013) proposed a model-based method by considering both disturbances and uncertainties, for detecting optical fiber sensor fault in aero-engine linear discrete time invariant (LDTI) system. The performance was demonstrated by applying the design procedure to a non-linear gas turbine model.

2.3.2. *Data-driven approach*

This approach involves training a machine learning model on large datasets to learn the patterns and relationships between input and output variables. The model can then be used to make predictions on new data. It does not require prior knowledge or assumptions about the data and can capture complex relationships between variables. Examples of data-driven techniques include neural networks, support vector machines, decision trees and others. (Yin & Wang, 2013) introduced a robust data-driven fault detection method which is based on a standard residual generation and decision logic structure for a wind turbine benchmark. The performance and effectiveness of their proposed scheme was demonstrated with simulation

result. (Rahimilarki et al., 2020) proposed a neural network sensor fault diagnosis approach for sensors in industrial systems that are prone to faults or malfunctions due to aging or accidents.

2.3.3. *Signature-based approach*

This approach uses a predefined signature or pattern of the sensor data to detect sensor faults. If the measured sensor data deviates from the expected signature, it is indicative of a sensor fault. An example of this approach is the use of a threshold-based detector to detect sensor faults. The detector compares the current sensor reading with a predefined threshold value. If the sensor reading exceeds the threshold, it would be an indicative of a fault. (Najafi Amin et al., 2019) performed a comparison of acoustic and vibration signature-based methods for detecting faults inside the bearings by using both time and frequency-based fault indicators i.e. RMS, kurtosis and envelope analysis for investigating the condition of the system and showed that acoustic signature-based methods can detect the system's fault from close distance, still detected some bearing conditions even for relatively far distances.

2.3.4. *Statistical approach*

This approach involves analyzing the statistical properties of the sensor data, such as mean, variance, standard deviation and correlation, to detect anomalies or outliers that indicate sensor faults. Statistical methods such as hypothesis testing, regression analysis, and clustering are used to identify these anomalies. (Abouelela et al., 2000) addressed the problem of textile quality control by implementing and integrating a set of statistical modules to detect faulty textiles. The modules were divided into preprocessing modules which consisted of normalization and mean filtering and processing modules which consisted of smoothing and variance. The algorithms were experimented with a set of images that consisted fault free and faulty textiles images. (Srimani et al., 2016) proposed a statistical method that maps faults to a statistical metric, the Bhattacharyya distance, which is calculated from the probability density function in order to detect parametric faults in linear and weakly non-linear analog circuits.

2.3.5. *Hybrid approach*

This approach involves combining two or more of the above approaches to improve the accuracy and robustness of the fault detection system. For example, a hybrid approach could combine a model-based approach with statistical approach to detect faults in a complex system.

2.4. Application of Sensor Fault Prediction

Sensor fault prediction is the process of using data analysis and machine learning to predict when sensors may fail or become faulty. This can be helpful in various industries to anticipate and prevent equipment failures, reduce downtime, and improve overall operational efficiency. A few of the major industries where sensor fault prediction can be applied include:

Manufacturing: sensors are extensively used to monitor and control production process. By predicting sensor failures in advance, manufacturers can proactively replace faulty sensors before they cause any issues, leading to increased efficiency and productivity.

Energy: sensors are used to monitor equipment such as turbines, generators and transformer. By predicting when sensors may fail, energy companies can schedule maintenance and repairs in advance, reducing downtime and improve overall equipment reliability.

Transportation: sensor fault prediction can be used to anticipate when sensors may fail, reducing the risk of accidents and improving overall safety. For example, again in aviation, sensors monitor engine performance, and predicting sensor failures can help airlines to avoid costly delays and cancellations.

Healthcare: sensors are used for monitoring patient vital signs, such as heart rate, blood pressure, and oxygen levels. Sensor fault prediction can help ensure the accuracy of these readings, reducing the risk of misdiagnosis or improper treatment.

Agriculture: sensors are increasing being used to monitor soil moisture, temperature and other environmental factors. Sensor fault prediction can ensure the accurate readings of the sensors thereby allowing farmers to make informed decisions about irrigation, fertilization, and other farming practices. Also reduce the risk of crop damage or yield loss.

2.4.1. *Machine learning for sensor fault prediction*

Machine learning is increasingly being used for sensor fault detection, prediction and diagnosing due to its ability to automatically learn patterns from data and identify anomalies that may indicate sensor faults. Machine algorithms can be trained on datasets of sensor measurements to identify patterns that may be indicative of sensor faults. Once trained, the algorithms can be used to predict sensor fault. Machine learning algorithms have several advantages of traditional approaches such as rule based or statistical methods. Machine

learning algorithms can automatically learn patterns from data, making them more adaptable to changing conditions and new types of faults. Additionally, machine learning can identify faults that may be difficult to detect using traditional methods, such as faults that occur intermittently or in complex systems.

There are several machine learning techniques that can be used for sensors:

Supervised: in this approach, the algorithm is trained using labeled data, where the sensor readings are labeled as either faulty or non-faulty. The algorithm then learns to classify new sensor readings as either faulty or non-faulty based on the patterns it has learned from the training data. Supervised learning can be further divided into classification and regression[4]. Classification is used when the output variable is categorical i.e., with 2 or more classes. Regression is used when the output variable is a real or continuous value. Some common algorithms used for supervised learning include logistic regression, which is often used for binary classification tasks. Logistic regression works by modelling the relationship between the input features (sensor readings) and the output variable (faulty or non-faulty). Support vector machine which is also used for binary classification. The algorithm works by finding a hyperplane that separates the faulty and non-faulty sensor readings. Although these algorithms are designed for binary classification, with a bit of modification they can be used for multi-class classification. This can be achieved by two methods: *One-Vs-Rest,* which is a strategy that works by fitting one binary classification model for each class vs all other classes. The second method, *One-Vs.-One,* works by fitting one binary classification model for each pair of classes.

Unsupervised: in this approach, the algorithm is not given any labeled data, but instead learns to identify patterns and anomalies in the sensor data on its own. This can be useful in cases where the specific faults are unknown or there are many different types of faults that can occur. Unsupervised learning can be further grouped into clustering and association. Clustering deals with finding a structure or cluster in a collection of unlabeled data. In simple terms, it is grouping or organizing objects that are similar in some way into a group or cluster. There are various similarity measures which can be used for grouping. For example, for vectors, *cosine distance* can be used. For sets, *Jaccard distance* and for points, *Euclidean*

[4] https://www.simplilearn.com/tutorials/machine-learning-tutorial/supervised-and-unsupervised-learning

distance can be used. Some common algorithms used for unsupervised learning include k-means clustering, which groups similar sensor readings together based on their distance from each other in a high dimensional feature space. Anomaly detection algorithms such as Isolation Forest, identify sensor readings that are significantly different from the rest of the data, which may indicate a fault. Association is a rule-based machine learning to discover the chances of probabilities of two events co-occurring in a given collection. The Apriori algorithm which works by first identifying the most frequent itemsets in a dataset and then use these frequent itemsets to generate association rule is one of the most widely used association algorithm. A good real-life example of this is the *market basket analysis*. This is a problem where in it is determined that if you buy a certain group of items, whether you are less or more likely to buy another group of items. This is especially useful in supermarkets. Other algorithms such as FP-Growth and ECLAT algorithms.

Semi-supervised: this approach combines both supervised and unsupervised learning by using a small amount of labeled data to guide the algorithm's learning process, while also allowing it to learn on its own from the larger amount of unlabeled data. Some common algorithms used for semi-supervised learning include: *Self-training,* which involves training a model on the labeled data, and then re-training the model on the combined labeled and pseudo-labeled data. Another approach is the *Co-training,* which involves training two separate models on different sets of features, and then using the predictions from each model to label the unlabeled data. Good examples of this include fraud detection, where a company say with 10 million users analyzes five percent of all transactions to classify as fraudulent or not, while the rest of the data isn't labeled as fraud or non-fraud. Web content classification, where billions of websites produce billions of contents, labeling would take huge resource and so variations of semi-supervised learning are used. Other examples also include text document classification and so on. So, generally, supervised, semi-supervised and unsupervised approaches would look like this.

Figure 2-2. Machine learning categories

Reinforcement: Reinforcement learning is a type of machine learning that does not fall under any of the above categories. In fact, this learning does not require any labeled data or training set. It involves an agent learning to interact with an environment in order to maximize a cumulative reward signal. It is basically a trial-and-error approach where feedback is received in the form of rewards or penalties for actions. The agent adjusts its behavior based on the feedback to maximize the reward and minimize the penalty.

The reinforcement learning process can be broken into several key components:

Environment: the external system with which the agent interacts. This can be a physical system, a virtual environment, or any other system that can provide feedback to the agent.

State: this represents the current state of the environment. The agent uses this information to make decisions about what possible action to take next.

Action: the decision made by the agent based on the current state of the environment. The action will have an impact on the environment.

Reward: the feedback received by the agent based on the action taken. This can be a positive reward or negative reward, and it is how the agent learns how to behave in the environment.

Policy: the strategy used by the agent to select the best actions based on the current state of the environment and the feed-back gained. The goal is to learn an optimal policy that maximizes the cumulative reward.

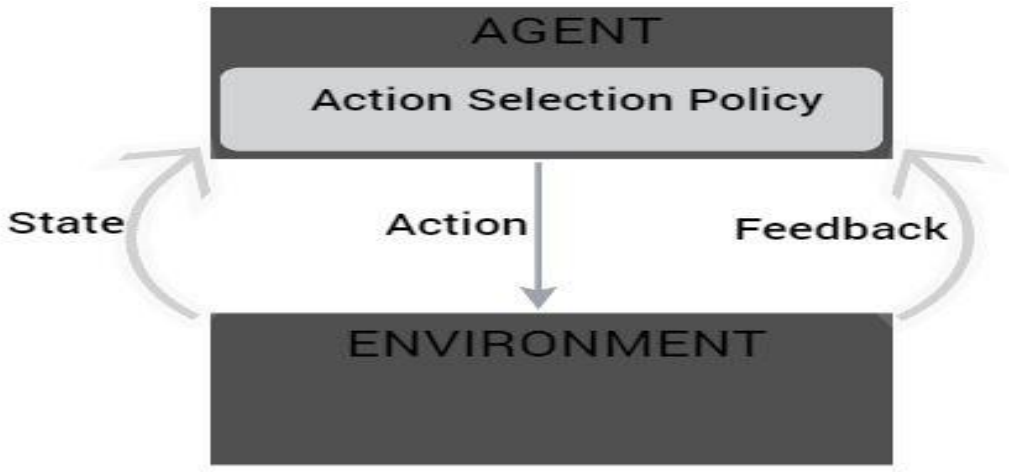


Figure 2-3. Reinforcement learning

2.4.2 *Deep learning for sensor fault prediction*

Deep learning is a subset of machine learning that uses neural networks with multiple layers to learn and extract features from complex data. While traditional machine learning techniques rely on manual engineered features and simple models, deep learning can automatically learn complex patterns and relationships in data, making it a powerful tool for a variety of applications. One of the main reasons deep learning is preferred over traditional machine learning is its ability to handle complex and high-dimensional data such as images, audio and text. Deep learning has shown great success in areas such as image recognition and NLP applications such as language translation, text classification, and speech recognition.

In sensor fault prediction, deep learning has been shown to be particularly effective because they can learn representations of the sensor data that capture both local and global features, and can automatically extract relevant features from raw sensor data. There are many advantages to using deep learning such as *higher accuracy* because they are able to learn more complex patterns in the data, and can capture subtle relationships between different variables that may be difficult for other methods to detect. *Better scalability* as deep learning algorithms can be scaled up to handle large amounts of data. This can also allow for the use of more complex models which can improve the accuracy of the predictions. *Flexibility* as deep learning algorithms are very flexible and can be adapted to a wide range of sensor types and applications. They can also be combined with other machine learning techniques, such as reinforcement or transfer learning, to further improve their performance. Some of the most commonly used deep learning techniques include:

Convolutional Neural Networks (CNNs): these are deep learning models designed specifically for image processing tasks. They use filters that convolve over the input image to extract important features and learn hierarchical representations of the image. They are particularly known for finding patterns in images to recognize objects, categories and classes. They have also been shown to be effective in dealing with classifying audio, time-series and signal data. A CNN is composed of 3 layers. The input layer, an output layer and a hidden layer which is usually more than one[5]. Overall, the layers alter the data through different operations with the intent of learning specific features of the data. The basic architecture of CNN consists of a convolution tool that separates and identifies distinct features of an image, through a process known as feature extraction and a fully connected layer that predicts the image's class i.e., performs classification, based on the features retrieved i.e., output of the convolution process.

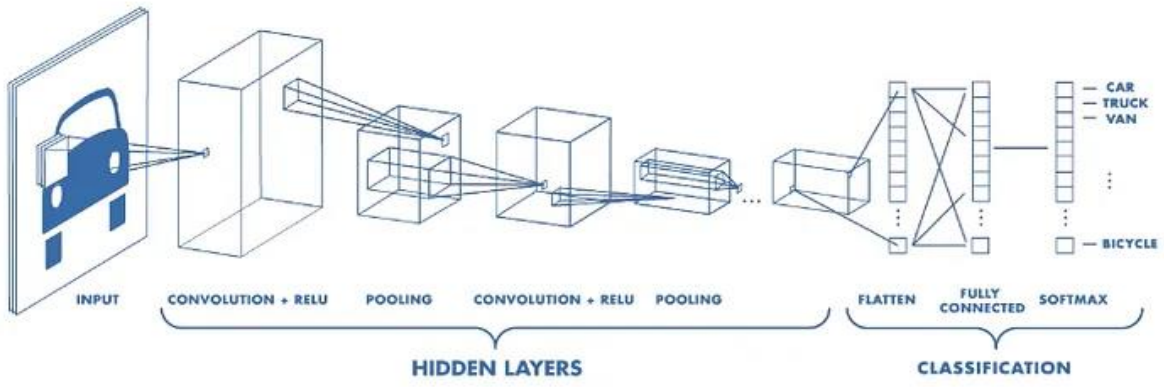


Figure 2-4. CNN architecture.[6]

Although CNN is mostly popular in computer vision and image processing tasks, it has also shown great potential in handling time-series data (Wibawa et al., 2022). The major difference between image processing or computer vision problems and time-series is that computer

[5] https://www.analyticsvidhya.com/blog/2022/03/basic-introduction-to-convolutional-neural-network-in-deep-learning/#:~:text=What%20is%20CNN%3F,is%20being%20known%20as%20Convolution.
[6] https://www.mathworks.com/discovery/convolutional-neural-network-matlab.html#:~:text=A%20convolutional%20neural%20network%20(CNN,%2Dseries%2C%20and%20signal%20data.

vision uses image matrix while time-series uses 1D array (Yamashita et al., 2018). So CNN can be adapted for tabular data by using 1D convolutional layers.

Recurrent Neural Networks (RNNs): these are deep learning models that can process sequential data, such as time-series or text-data, by maintaining a hidden state that captures information about the previous inputs in the sequence. Each input word has the same weight in the RNN, and the parameters are shared between different parts. In traditional neural networks, there is no concept of using previous output as input for the current step, and input and output are independent of each other. However, in cases such as prediction and forecasting, for example, predicting the next word in a sentence or forecasting future sales, the previous data is required, and hence there is a need to remember the previous input (Hiransha et al., 2018). And so, unlike multilayer perceptron, RNN takes input from two sources. One of the inputs comes from the past and the other comes from the present. RNN can only run for a limited number of steps because of problems with vanishing gradients. As a result, various architectures such as Long Short-Term Memory (LSTM), Gated Recurrent Unit (GRU), Bidirectional LSTM, Stacked LSTM, and LSTM with attention method have been developed to overcome the limitations of RNN.[7]

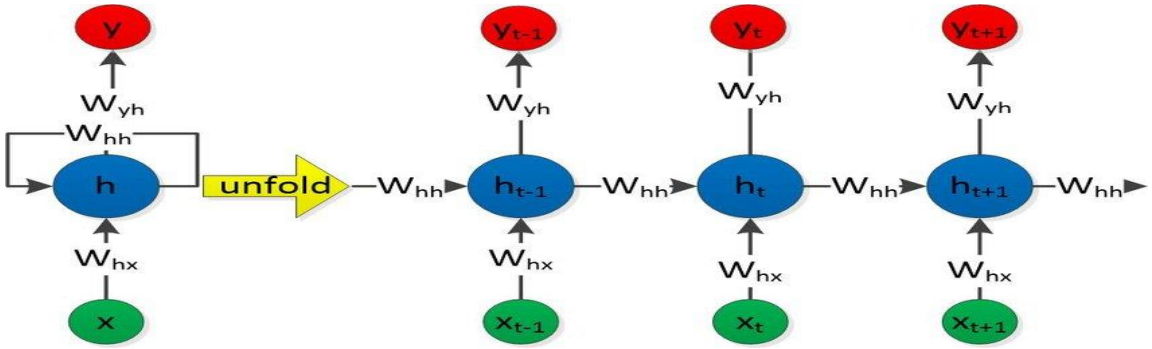


Figure 2-5. RNN architecture (Zheng et al., 2017)

Long Short-Term Memory (LSTM): these are a type of RNN that can capture long-term dependencies in sequential data by using gates that control the flow of information into and out of the hidden state. The key to LSTM is the cell state. The cell state is used to maintain long-term memory. LSTMs add or remove information from the cell state using a carefully

[7] https://towardsdatascience.com/lstm-networks-a-detailed-explanation-8fae6aefc7f9
https://machinelearningmastery.com/crash-course-recurrent-neural-networks-deep-learning/

regulated structures called gates. There are 3 gates in LSTM that control the flow of information in and out of the memory cell i.e., the *forget gate, input gate* and *output gate.*

The forget gate decides whether we should maintain information from the previous time-step or forget it. The equation of the forget gate is:

- $\mathbf{ft = \sigma(X_t * U_f + H_{t-1} * W_f)}$

where X_t is the input to the current time-stamp

U_f is the weight associated with the input

H_{t-1} is the hidden state of the previous time-step

W_f is the weight matrix associated with the hidden state

The output of these products is then passed though a sigmoid function (**σ)** to make it a number between 0 and 1 where 0 represents forget everything and 1 represents keep everything.

Not all kept information is equally important or relevant. So, a mechanism of quantifying the importance of the new information is required. This is where the input gate comes in. The new information is calculated by $N_t = \tanh(X_t * U_c + H_{t-1} * W_c)$. Because of the tanh activation function, the value of the new information will be between -1 and 1. If a negative value is obtained, the information is subtracted from the cell state otherwise it is added to the cell state at the current timestamp.

To achieve the final output, the SoftMax activation function is applied to the current hidden state Ht. The current hidden state Ht can be calculated using the output gate and tanh of the updated cell i.e., $H_t = O_t * \tanh$. The equation of the output gate (O_t) is pretty similar to the other two gates. $O_t = \sigma(X_t * U_o + H_{t-1} * W_o)$. The SoftMax activation is applied to normalize the outputs to a probability distribution over the predicted classes and so the token with the maximum score will be predicted as the output. A more intuitive diagram of the networks looks like:

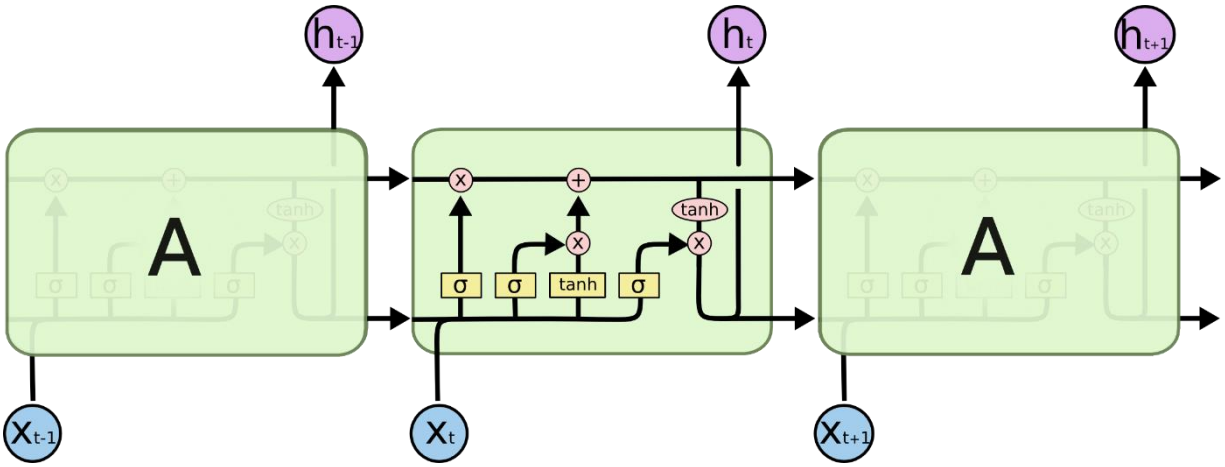


Figure 2-6. LSTM architecture [8]

AutoEncoders: these are deep learning models that can learn compressed representation of the input data by encoding it into a lower-dimensional space and then decoding it back into original input space. They are often used for data compression, denoising, and anomaly detection.

2.5. Time-Series Analysis and Forecasting

Time series analysis is a technique used to analyze and interpret data points collected over time. This typically involves analyzing seasonality, trends, cyclical fluctuations, and irregular variations to identify patterns and relationships in the historical data to model the behavior of the data over time. Forecasting, on the other hand, involves using the information learned from timeseries analysis to predict future values of the variable being studied. This can be done using various statistical and machine learning techniques such as regression, neural networks and deep learning. Time series analysis and forecasting are used in a wide range of fields such as economics, finance, meteorology, and healthcare to name a few. Accurate forecasting can help organizations make better decisions and prepare for future changes, leading to improved outcomes and increased success.

2.5.1. Univariate Time-series Analysis

Univariate time-series analysis is the study of a single variable that is observed over time. This type of analysis is used for studying and predicting the behavior of a single variable, such as stock prices, temperature, or number of customers at a store. The problems under these

[8] https://colah.github.io/posts/2015-08-Understanding-LSTMs/

categories are comprised of a single series of observations and a model is required to learn from the series of past observations. Once the patterns and trends in the historical data is analyzed and learned, univariate timeseries forecasting can be performed using statistical and machine learning techniques, such as exponential smoothing, ARIMA and Prophet.

2.5.2. Multivariate Time-series Analysis

Multivariate time series analysis involves the study of multiple variables that are observed over time. This type of analysis is useful for studying and the relationships between multiple variables, such as stock prices, interest rates, and inflation. The focus in this analysis is on understanding the relationships and dependencies between the variables in the historical data. Various statistical and machine learning techniques such as Vector Autoregression (VAR) and Recurrent Neural Networks (RNN) can be used for forecasting.

In most cases, time-series forecasting is discussed in the case where only one-step prediction is required, that is, predicting the observation (value) at the next time step based on past observations. However, in most real-life situations forecasting a single timestep becomes too narrow for many problems, i.e., predicting many timesteps ahead becomes necessary. This is called multi-step time-series forecasting. Predicting many time-steps ahead has important implications. It reduces long-term uncertainty, thereby enabling better planning. There are many approaches to multi step forecasting discussed throughout the literature. Below we'll discuss the two most common approaches used.

2.5.3. Recursive Multi-Step Forecasting

This approach involves predicting the next value in the time series and then using that predicted value as an input to predict the next future time step. This process is repeated until the desired number of future values has been predicted. This approach allows for corrections to be made as new information becomes available. The downside to this approach is the error generated in each step is propagated to the subsequent steps. If we are using it for long-term forecasts, the errors will eventually accumulate and degrade the performance as the time-steps increase (Marcellino et al., 2006)(Taieb & Hyndman, 2022).

2.5.4. Direct Multi-Step Forecasting

This approach builds a separate model for each time-step, i.e., if we wanted to forecast two time-steps in the future, we would build a model for predicting time-step t1 and a separate

model for predicting time-step t2. While this approach does avoid the error propagation situation we saw in recursive multi-step forecasting because there's no iteration on any model, building separate models for each time-step is computationally expensive and is a maintenance burden (Taieb & Hyndman, 2022). Also, we cannot model the dependencies between the different time-steps, i.e., t1 and t2, as they are being predicted by different models (Taieb & Hyndman, 2022).

2.6. Multi-class Classification

Multi-class classification is a type of machine learning task where the goal is to classify input data into one of three or more distinct classes or categories. In this scenario, each data instance belongs to only one class. There are two main approaches to multi-class classification. Direct /native multi-class classification and binary transformation (class binarization)

2.6.1. Direct Multi-class Classification

In direct multi-class classification, the classification algorithm is designed to handle multiple classes natively(directly), without any explicit transformation or modification. The algorithm assigns a single class label to each input instance directly. Some algorithms that inherently support multi-class classification include decision trees, random forests, k-nearest neighbors (KNN), and naive Bayes. Another common example of a direct multi-class classification algorithm is the Softmax regression (also known as multinomial logistic regression) which extends the logistic regression to handle multiple classes by using the Softmax function.

2.6.2. Binary Transformation

Class binarization, also known as binary transformation, involves transforming a multi-class classification problem into multiple binary classification subproblems. Instead of directly predicting the multi-class label, a set of binary classifiers is trained to distinguish between each class and the rest of the classes. There are two common strategies for class binarization: one-vs-all (OvA) and one-vs-one (OvO).

One-vs-All (OvA): In the OvA strategy, a binary classifier is trained for each class, treating it as the positive class and all other classes as the negative class. During prediction, each binary classifier is used to classify a new instance, and the class with the highest confidence or probability is assigned as the predicted class.

One-vs-One (OvO): In the OvO strategy, a binary classifier is trained for every pair of classes. For N classes, N(N-1)/2 classifiers are trained. Each binary classifier is trained on a subset of the data that includes only the instances from the two corresponding classes. During prediction, each binary classifier votes for its predicted class, and the class with the highest number of votes is assigned as the predicted class.

2.7. Baseline Work

Due to lack of publicly available benchmark sensor datasets, it is common among researchers to each annotate their datasets themselves leading to inconsistencies as a result of different methods of annotations. In order to address this (De Bruijn et al., 2016) presented a standardized approach to annotating benchmark datasets and algorithms for injecting faults. Many researchers have used the standardized approach discussed in (De Bruijn et al., 2016) to inject artificial faults and annotate their datasets to evaluate their models (Wang et al., 2019) (Titouna et al., 2019) (Mishra & Mohanty, 2022). In this research, we use as a baseline the work done by (Emperuman & Chandrasekaran, 2020). The authors prepared the faulty dataset by introducing four faults i.e., spike, random, drift and bias at the rate of 10%, 20%, 30%, 40% and 50% to the raw Intel Lab dataset using the injection algorithms described in (De Bruijn et al., 2016). By using a continuous density hidden Markov model (CDHMM), their highest classification accuracy was for the 10% injection rate with average accuracy of 97.5% (average of the 4 faults) and their lowest was 50% injection rate with average accuracy of 87%. The 20% injection rate average accuracy was 95%. Since our dataset is injected with a 20% and 21% injection rate, we used their 20% injection rate performance to validate the performance of our model.

2.8. Related Works

Many researches have been conducted to address the issue of sensor faults using the different approaches such as model-based, data-driven, signature-based, statistical or by combining two or more of the above.

(Manzoni et al., 2021) employed a model-based fault detection to detect Continuous Glucose Monitoring (CGM) sensor's fault in an artificial pancreas. A one-step ahead Kalman predictor is used to predict the expected blood glucose level in the normal conditions, accounting also for the meals, carbohydrate contents, and insulin injected by the CSII pump. This predicted

value is then compared against the measured values by the CGM. Large inconsistencies between the measured CGM values and predicted values of the blood glucose level would suggest the presence of faults.

(Jihani et al., 2023) employed a parity space approach to detect and isolate faults in WSN. A mathematical model was developed based on the instantaneous redundant measurement of two sensor nodes planted in the soil with the same depth. A residual signal (parity vector) is then generated based on the mathematical model. The signal is then used to detect the presence of fault. When there are no faults, the residual signal equals to zero or near to zero and when there is a presence of fault it deviates from zero.

(Yan et al., 2021) employed a novel model to for detecting minor soft faults of sensors in an air conditioning system. The authors used kernel principal component analysis for feature extraction and dimensionality reduction, then the data was passed to a double layer bidirectional long-short term (DL-BiLSTM) after being converted into sequences with sliding window. The residual which is generated by comparing the output of the DL-BiLSTM with the actual value from the supply air temperature sensor is used to detect the minor faults of the sensors. According to the authors their experiment result showed that their method (KPCA-DL-BiLSTM) was 43% higher than that of KPCA and 18.33% higher than LSTM under 10% drift deviation fault.

(Alwan et al., 2022) proposed a time-series clustering technique to address the limitations of predictive analysis in detecting long-segmental faults in sensor nodes in large scale cyber physical systems. They also compared feature-based time-series clustering and shape-based time-series clustering and found that feature-based time-series clustering is a more efficient long-segmental outliers detection mechanisms.

(Zhao et al., 2022) employed sliding window approach to detect two incipient faults, i.e., constant bias and precision degradation, in industrial processes. The singular value of each window is calculated. The control limits are determined by empirical method using the mahalanobis distance, where each singular value corresponds to a control limit. If all the singular values in a sliding window are not greater than corresponding control limits, then the sample is considered normal.

(Uppal et al., 2022) developed an early fault prediction model for an IoT environment that uses IoT-based sensors, cloud and ML algorithms. The model was evaluated against four algorithms, i.e., decision tree, k-nearest neighbor, gaussian naïve bayes and random forest. According to the authors Random Forest showed the highest performance on the given dataset, with a classification accuracy of 94.25%.

(Liu et al., 2022) proposed a deep learning approach for sensor fault self-diagnosing scheme for a wind turbine blade that does not depend on labeled fault data. First a sensor data prediction model was built by mining the inherent relevance between the sensors, and then the residual between the predicted value and the actual value is compared against the control limit to detect the presence of faults. According to the authors, the experimental results of the model showed a good prediction (RMSE = 0.001154, MAE = 0.000214, and R2 = 0.999993) and fault diagnosis performance (a recall of up to 98.43%, an accuracy of up to 98.58%, and a precision of up to 90.01%).

(Wahid et al., 2022) proposed a predictive maintenance based on convolutional neural network and long short-term memory for machine failures in industrial plants and factories. LSTM was used to analyze the relationships among the different timeseries data and 1D CNN was used for the effective extraction of high-level features. The authors experimented with CNN, LSTM and CNN-LSTM models for the prediction and found that CNN-LSTM provided the most reliable and highest prediction accuracy over the others.

(Uppal et al., 2021) proposed an architecture for monitoring each and every office appliance connected via IoT using machine learning. Data from the appliances is collected by IoT and is sent to the ML algorithms. The ML algorithm performs the classification process of the faults. The information is then passed to a recommendation system which alerts the user on deviation or abnormal working of any appliance and whether the device needs repairing or replacing.

(Safavi et al., 2021) proposed a novel method for predicting the health status of the electronics sensors of autonomous vehicles. 1D CNN was used to fuse feature extraction and learning process of the raw sensor signals. For fault identification and isolation, a separate feature extraction block was used and to identify the type of fault and recognize the faulty sensors multi-class DNN was used.

Reference	Methods and algorithms	Results
(Manzoni et al., 2021)	One step Kalman predictor with correlogram test and residual punctual value analysis	Residual value analysis showed good performance in terms of sensitivity and FP/day, while correlogram test, although weak in short and minor amplitude fault, is better in longer and deeper faults.
(Jihani et al., 2023)	Parity space approach with direct redundancy	Injected faults were immediately detected and isolated
(Yan et al., 2021)	KPCA and double layer Bi-LSTM	10% injection= 97.33%ACC, 99.00% Recall, 97.07% precision. 20% injection= 96.67% ACC, 99.5% Recall ,95.67% precision
(Alwan et al., 2022)	Dynamic Time Warping (DTW), K-Shape timeseries clustering and characteristic based timeseries clustering	Feature-based techniques more efficient than shape-based techniques for long-segmental fault detection
(Zhao et al., 2022)	Sliding window-based algorithm that monitors singular value	95.10% and 95.0% using numerical evaluation and 98.57% and 98.50% using CSTR process for bias and degradation respectively
(Uppal et al., 2022)	DT, k-nearest neighbor, Gaussian NB and RF	RF performed best with accuracy of 94.25%
(Liu et al., 2022)	RF, BP and CNN	Prediction: RMSE = 0.001154, MAE = 0.000214, and R2 = 0.999993 Diagnosis: Recall 98.43%, Accuracy 98.58%, and a precision of 90.01%
(Safavi et al., 2021)	1D CNN, multi-class DNN	Deep learning outperformed SVM
(Biddle & Fallah, 2021)	SVM for detection and identification and predictive algorithm for prediction	detection and identification accuracies of 94.94% and 97.01%, respectively and prediction accuracy 75.35%

Table 2-1. Summary of related works

2.9. Summary of Related Works

In summary, the existing studies have suggested different approaches such as model-based, signature based and data-driven approaches for detecting sensor faults and potential failures. The model-based approach uses a mathematical model (set of equations and rules) of the system to estimate the true value of a sensor reading and detect sensor faults by comparing the predicted and measured values (Gao et al., 2015) (Habibi et al., 2019). The signature-based approach compares the current sensor reading with a pre-defined threshold value. If the reading exceeds the threshold, it's an indicative of a fault (Clouqueur et al., 2001). The data-driven approach analyzes the statistical properties of the sensor data to learn the underlying patterns and relationships between the sensor reading and the occurrence of faults (Subbaraj & Kannapiran, 2010). However, the existing studies still have drawbacks in the following sense:

- The existing literature mainly focuses on fault detection rather than prediction. While detecting a fault is useful it only addresses the problem after it has occurred, meaning that actions are taken after the fault has already impacted the system, potentially leading to negative consequences.
- The literature is limited in the number of faults considered. Many studies focus on specific faults and so they cannot be generalized to other faults. This is a problem if we consider the deployment or application of this systems in the real world because a real-world setting will likely exhibit multiple fault types at the same time.
- The existing studies that aim to predict multiple faults still require improvement in their accuracy.

CHAPTER THREE

RESEARCH METHODOLOGY

3.1. Introduction

The quality and reliability of IoT systems depend on the quality and reliability of sensors and the data fed into it by the sensors. Corrupt or faulty data from the sensors can compromise the IoT system making it unreliable, inaccurate and in some cases even leading to system failures. The ability to predict potential sensor faults before they manifest can facilitate proactive maintenance, minimize downtime, and enhance overall operational efficiency of the system.

In this research we aimed to anticipate potential sensor data faults by developing a 2-step hybrid deep learning models for the prediction and classification of sensor faults. By leveraging the power of deep learning algorithms with attention mechanisms, which have shown remarkable success in various fields, we created an effective and robust solution that can predict and classify sensor faults with high accuracy. To show the effectiveness of our proposed system we used the publicly available intel lab data that has been annotated and fault-inject by (De Bruijn et al., 2016). All the necessary preprocessing and feature-engineering was performed on the data before using it for training. We also compared our work against a baseline work and showed that our solution has improved accuracy.

The outcomes of this study have implications for various sectors, including manufacturing, agriculture, transportation, and healthcare, where the prediction of potential sensor faults can prevent catastrophic failures, reduce maintenance costs, and optimize system performance. Furthermore, the proposed hybrid deep learning models can serve as a foundation for future research in the field of predictive maintenance and fault diagnosis.

The subsequent sections of this chapter will present the research design, data collection methods, data preprocessing and feature engineering tasks performed, model selection, overview of the dataset, design and development tools used i.e., software and hardware and evaluation metrics used to achieve the research objectives.

3.2. Research design

The aim of this study is to predict faults in sensors and classify their types or classes. This involves training and testing a deep learning model on a datasets of sensor readings to predict or forecast future time-step values of the sensors. The forecasted values will then be fed to a second model which has been trained to classify sensor readings as normal or one of the 5 classes we consider in this study. The study also involves using statistical measures to evaluate the performance of the models such as RMSE, MAE, accuracy, confusion matrices and others. So, our study follows the experimental research design type. This design type allows us to establish the causal relationship between the independent variables, in our case the models, and the dependent variables, in our case, the prediction and classification evaluation metrics. So, by manipulating the independent variables (the models), we aim to make the prediction and classification performance higher.

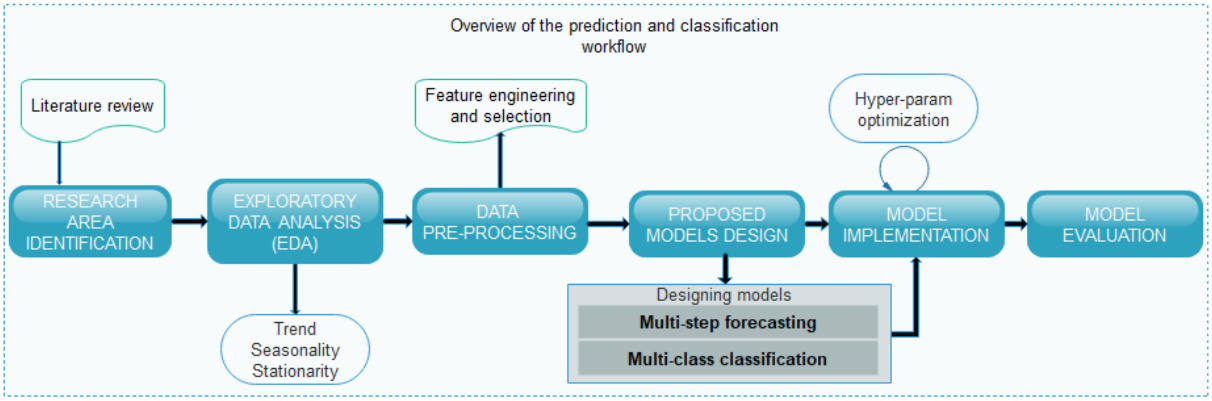


Figure 3-1. Research methodology workflow

3.3. Overview of the dataset

Due to lack of publicly available labeled datasets based on real-world sensor data where all the data points have been annotated with the ground truths and contains faults (natural or artificial) that reflects various abnormalities, it is a common practice among researchers to artificially inject fault signals to the sensor data in order to evaluate their system using different algorithms and fault models (De Bruijn et al., 2016). (Wang et al., 2019) using the Intel Berkeley Research Lab dataset (nodes 1,2,33,35 and 37) injected artificial outliers to the normal data to train and evaluate their proposed framework. The outliers consisted 3% of the data and the distribution was random. (Emperuman & Chandrasekaran, 2020) also introduced bias, drift, random and spike artificial faults to the raw data using the algorithm described in

(De Bruijn et al., 2016). (Mishra & Mohanty, 2022) injected intermittent fault signals to the Intel lab temperature and light dataset to test the performance of various machine learning classifiers for intermittent fault diagnosis.

The benchmark datasets used in this paper is prepared by (De Bruijn et al., 2016). The authors try to address the issues of inconsistencies between datasets that is caused by researches having to annotate the datasets by themselves in order to obtain ground truth. Since researchers use different methods of annotating the dataset thus creating different ground truths, and because annotated datasets are generally not shared, this leads to inconsistencies even when using the same raw data. The authors presented a standardized approach (methods and algorithms) to annotate benchmarks and inject faults into clean datasets based on a generic fault model.

3.3.1. Dataset source and description

The authors (De Bruijn et al., 2016) published 3 benchmark datasets for fault detection and classification. The raw data was collected from 3 publicly available sources, i.e., the Intel Berkeley Research Lab[9] (available at: http://db.csail.mit.edu/labdata/labdata.html), SensorScope[10](available at: https://zenodo.org/record/2654726#.ZDZsUuZBzIU) and SmartSantander[11].

The dataset used in this study contains 10 sensors from the intel lab (temperature). Each sensor has a clean set annotated with the ground truths and 4 sets injected with 4 artificial faults they surveyed from the literature, based on domain expertise and algorithmic methods described in their paper. The dataset is a timeseries data with features timestamp, mote_id, temperature, and has_fault_type. The has_fault_type feature is encoded as 0 for no fault, 1 for random, 4 for bias, 8 for drift, and 16 for poly drift. The authors have discussed the implementation details of the algorithms and parameters used for injecting the faults in their paper.

A sample data from the Intel sensor 1 run in visual studio code:

[9] https://www.intel.com/content/www/us/en/research/overview.html
[10] https://www.epfl.ch/research/
[11] https://smartsantander.eu/

data

0.1s

	timestamp	mote_id	temperature	has_fault_type
0	2004-02-29 00:00:00	1	19.2600	0
1	2004-02-29 00:00:30	1	19.2500	0
2	2004-02-29 00:01:00	1	19.2500	0
3	2004-02-29 00:01:30	1	19.2436	0
4	2004-02-29 00:02:00	1	19.2400	0
...	...	...	...	...
16123	2004-03-05 14:21:30	45	22.4384	0
16124	2004-03-05 14:22:00	45	22.4188	0
16125	2004-03-05 14:22:30	45	22.4384	0
16126	2004-03-05 14:23:00	45	22.4580	0
16127	2004-03-05 14:23:30	45	22.4900	0

161280 rows × 4 columns

Figure 3-2. Sample data

To summarize the features with their data type and description:

No.	Feature	Data type	Description
1.	Timestamp	Object	The time the value is recorded
2.	Mote_id	Int64	Sensor identifier
3.	Temperature	Float64	The measured temperature value
4.	Has_fault_type	Int64	The type of fault. Encoded in numbers.

Table 3-1. Dataset description

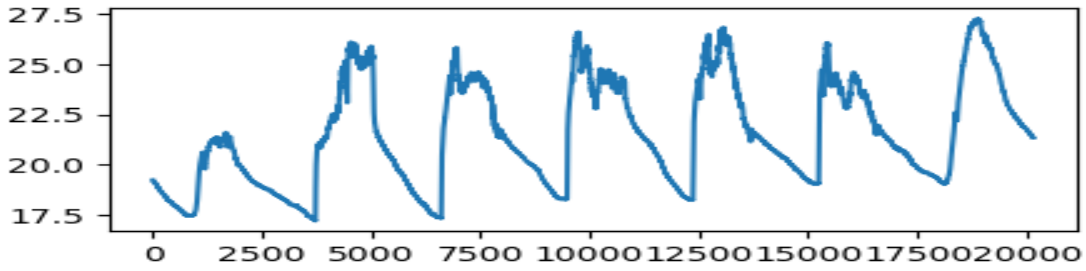
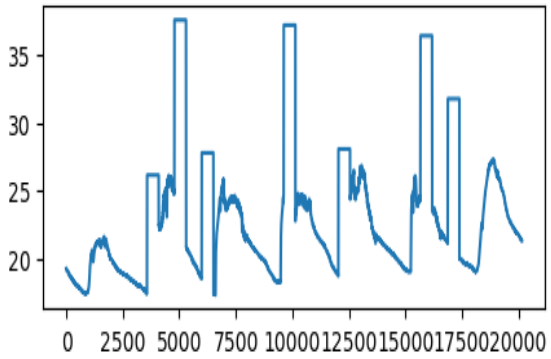
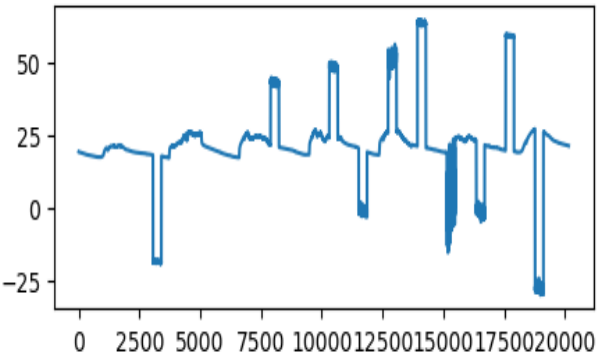


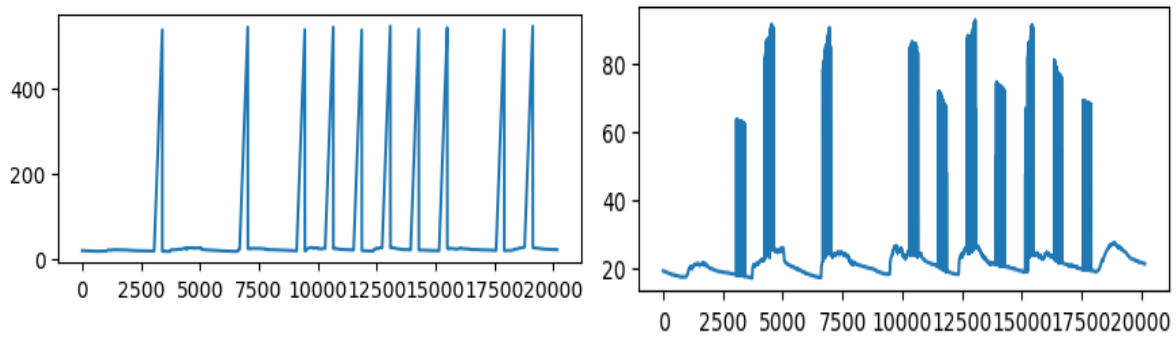
Figure 3-3. Cleaned temperature data from node 1



A. with 21% bias fault



B. with 20% drift fault



C. with 20% Polydrift *D. with 20% random*

Figure 3-4. Fault injected sets with their injection rate

Figure 3-4 shows the injection rate used. The main reason for the 20% injection rate is to create a realistic fault scenario. In real-world applications, sensor faults are relatively rare events as opposed to the normal condition, and having a small proportion of such instances in the dataset may not adequately represent the challenges faced during actual deployment. By injecting a higher rate of faults such as 20% in our case, the dataset is made more robust and capable, also this allows for a better evaluation and validation of the model's performance under different fault conditions, ensuring its effectiveness in real-world scenarios.

3.3.2. Data preprocessing

Data preprocessing is a crucial step in data analysis that involves transforming and preparing raw data into a format that is suitable for analysis. It ensures that the data is clean, consistent, and usable.

Some of the tasks we performed as part of data preprocessing includes:

- Data cleaning: one of the most common issues faced with datasets is missing values. Missing values are usually represented with the special value NaN. Missing values can be handled by inputting them with a specific value, or as we did in our case, impute them with special metrics like mean or median.
- Data integration: this involves combining data from multiple sources into a single data frame to obtain a comprehensive dataset. This is necessary in our case because our data is collected independently from independent sensors, and so the data from different sensors need to be integrated together to train a model.

- Data transformation: this involves converting the data into a suitable format for analysis. It may include normalizing, scaling, or converting data types. In our case we used MinMaxScaler to transform the data. We also converted the data type of the timestamp feature from object to datetime format.
- Imbalance handling: dataset imbalance occurs when the instance of one class is significantly higher or lower than other classes in classification. Highly imbalanced dataset can lead to biased model during training which can result in poor predictive performance especially on the minority classes. Techniques such as under-sampling, over-sampling and SMOTE are used to balance datasets.
- Stationarity: the data initially contained an increasing trend. Differencing operation was performed in order to remove the trend. Then adfuller test was then conducted to check the stationarity of the data. The test returned a p-value of 0.03 which is less than 0.05 implying the data is stationary.
- Resampling: in recursive multi-step forecasting, there is a tradeoff between number of timesteps to predict and the amount of error to tolerate. The longer the timesteps the higher the error. So, in order to forecast longer timesteps with as little error as possible, the sensor measurements were resampled from 30 seconds to 2 minutes. In order to minimize the loss of information and avoid undersampling, we followed a 2 -step process of resampling where we first down-sampled the data to 1-minute intervals by taking the average of every two consecutive 30-second data points. This reduces the frequency while still capturing the overall trends in the data. Then we applied spline interpolation technique to estimate the values at 2-minute intervals. Spline interpolation is more flexible and can capture more complex patterns in the data. It is especially useful when you want to preserve the shape and characteristics of the original data.

3.3.3. Feature engineering and selection

In the case of timeseries forecasting, where we try to leverage the power of analyzing historical data to forecast future timesteps, feature engineering and selection plays 2 major parts.

1. Generation of new features

The predictive power of any machine learning algorithm lies in its knowledge and understanding of the data. The more a model is able to understand the dynamics and relationship that exists between the instances or observations, the more it will be able to make better and accurate forecasts. Hence new features can be generated to give the model additional information and insight about the data to improve its performance. There are two major types of features that are generated from the original features for timeseries problems.

Time-based features: Time-based features are based on the time dimension of the data and can capture the temporal patterns and trends that are present in the time series. Examples include:

- Date/Time features: These features include the year, month, day of the week, hour, minute and second of the data point. These features can be used to capture weekly, monthly, yearly or seasonal trends.
- Moving averages: These are calculated by taking the average of the time series over a fixed window of time. For example, a 24-hour moving average of hourly temperature values could be used to capture the daily trend.
- Seasonal indicators: These features capture the seasonality of the timeseries. For example, if you are forecasting a temperature values, you can include this feature to capture the changes in temperature during summer or winter as this season greatly contribute to temperature changes. Other features such as holidays/events and weekends or not could all be used under different problem sets. For example, holidays can play major role in sales forecasting as sales tend to peak during holidays and so on.

Statistical features: Statistical features are based on the statistical properties of the data and can be used to capture trends and patterns that are not explicitly based on time.

- Rolling mean: This feature calculates the mean of, in our case, the sensor readings over a fixed window of time. This feature can help identify trends and patterns in the data.
- Rolling standard deviation: This feature calculates the standard deviation of the sensor readings over a fixed window of time. It can help detect changes in the variability or noise of the data.
- Rolling minimum and maximum: These features calculate the minimum and maximum values of the sensor readings over a fixed window of time. They can be used to identify extreme values or outliers in the data.

- ❖ Lag features: These features capture the historical behavior of the sensor readings. For example, lag-1 feature represents the sensor value at the previous time-step, lag-2 represents the value at two time-steps ago, and so on. They can help to detect patterns and trends in the data and capture the temporal dependencies.

It is also common to combine both time-based and statistical features and add them to the model as additional information for the model in order to build more robust models for time-series forecasting. By combining both types of features, we were able to capture both the temporal patterns and statistical properties of the data, leading to more accurate forecasts.

2. Framing the data into a supervised learning problem

The second important task performed is turning the timeseries dataset into a dataset suitable for supervised learning problem. Generally, classification and prediction/forecasting are problems covered by supervised learning. The way a supervised algorithm works is by training on a training dataset which is split into input variables (X) and an output variable (y), and the algorithm learns a mapping function from the input to the output and uses this function to classify new unseen data points or forecast new values[12]. This is achieved by using the values at the previous n number of timesteps as input and the current timestep as output. This method is called window sliding in some literature and in statistics it is called the lag method where the previous values are referred to as lag values.

An example with 2 previous lag values would look like:

		Input variables(X)		Output(y)
T0		T-2	T-1	T0
10		NaN	NaN	10
15		NaN	10	15
20	→	10	15	20
25		15	20	25
30		20	25	30
35		25	30	35

Figure 3-5. Sliding window for time-series

[12] https://machinelearningmastery.com/time-series-forecasting-supervised-learning/

Feature selection is the process of selecting a subset of relevant features from a larger set of features. This is because not all features are relevant or contribute equally for the specific predictive model we want to build. Feature selection allows us to reduce the irrelevant or less relevant and redundant features thereby simplifying the model and making it more interpretable. The choice of the feature selection will of course depend on the nature of the dataset, the algorithms used and on the objective of the model. Generally, feature selection can help reduce overfitting, improve accuracy and interpretability and also reduce computational costs. Some of the techniques for feature selection include:

1. Filter methods:

Filter methods select features based on their statistical properties, such as their correlation with the target variable and variance. These methods are independent of the machine learning algorithm used(Eseye et al., 2019). Some common filters include:

Correlation-based feature selection (CFS): This method ranks features based on their correlation with the target variable and the inter-correlation among themselves. It uses entropy-based metric to measure the relevance of each feature and selects the top-ranked features.

Chi-squared test: This method measures the dependence between each feature and the target variable by comparing the observed frequencies with the expected frequencies under the null hypothesis of independence. It selects the features with the highest chi-squared statistics.

Variance threshold: This method removes the features with low variance, assuming they are less informative and selects the features with variance above certain threshold

2. Wrapper methods:

Wrapper methods use a machine learning algorithm to evaluate the performance of a feature subset and select the best subset based on a performance metric such as accuracy or AUC.

Recursive feature elimination: This method starts with all the features and removes the least important feature iteratively until a stopping criterion is met, such as the desired number of features or the highest performance.

3. Embedded methods:

Embedded methods combine feature selection with model building process, and the model learns which features are most relevant during training.

Lasso regularization: This method adds a penalty term to the objective function of the model, which encourages sparsity and shrinks the coefficients of irrelevant features to zero. It selects features with non-zero coefficients.

Decision tree-based methods: This method uses decision-trees to recursively split the feature space and selects the features that provide the most information gain or Gini impurity reduction.

3.4. Model Selection

The goal of model selection is to choose a model that effectively captures the underlying patterns in the sensor data, accurately forecasts and classifies the faults. Since sensors are timeseries data, the ability to capture the temporal dependencies is the major criteria for the model selection. The ability of the model to generalize well on unseen data is the second criteria. Based on these criteria, we identified the following candidate models and evaluated each of them.

For the regression task:

Long Short-Term Memory (LSTM): LSTM networks are capable of capturing long-term dependencies in sequential data, making them suitable for time series analysis. Their recurrent architecture and memory cells enable them to retain and process historical sensor data effectively.

Bidirectional LSTM (BiLSTM): BiLSTM is an extension of the LSTM model that considers both past and future information by processing the input sequence in forward and backward directions. It captures the long-term dependencies and temporal patterns effectively, making it suitable for sensor fault prediction tasks.

Gated Recurrent Unit (GRU): Similar to LSTM, GRU is a type of recurrent neural network that also addresses the vanishing gradient problem. GRU has a simpler architecture with fewer gates than LSTM, resulting in faster training and better performance on smaller datasets. It can effectively capture temporal dependencies in sensor data.

CNN-LSTM: CNN-LSTM is a hybrid model that combines the strengths of Convolutional Neural Networks (CNNs) and LSTMs. CNNs are excellent at extracting spatial features, while LSTMs capture temporal dependencies. The CNN-LSTM model leverages CNNs to extract relevant spatial features from sensor data and passes them to LSTM layers to capture temporal patterns and make predictions.

CNN-GRU: Similar to CNN-LSTM, the CNN-GRU model combines CNNs with Gated Recurrent Units (GRUs). It leverages the spatial feature extraction capabilities of CNNs and the ability of GRUs to capture temporal dependencies. This model is particularly useful when both spatial and temporal information are crucial for sensor fault prediction.

CNN-BiLSTM: The CNN-BiLSTM model combines the strengths of CNNs and Bidirectional LSTMs. It utilizes CNNs to extract spatial features from sensor data and then feeds the extracted features into BiLSTM layers to capture temporal dependencies in both forward and backward directions. This model can effectively capture complex spatial and temporal patterns in sensor data.

For classification task:

CNN: CNNs are well-suited for fault classification tasks that involve spatial patterns in the sensor data. CNNs can automatically learn hierarchical representations of features through their convolutional and pooling layers. This allows them to effectively capture spatial dependencies and extract relevant features from the sensor data. In fault classification, CNNs can detect patterns indicative of specific fault types, such as distinctive shapes or spatial arrangements in sensor readings.

MLP: MLPs, also known as fully connected neural networks, are versatile models that can be used for fault classification tasks. MLPs consist of multiple layers of interconnected neurons, with each neuron being fully connected to the neurons in the adjacent layers. MLPs are known for their ability to learn complex nonlinear relationships in the data. In fault classification, MLPs can capture the non-spatial patterns and correlations present in the sensor data, allowing them to discriminate between different fault types based on these patterns.

CNN-MLP: The CNN-MLP hybrid model combines the strengths of both CNNs and MLPs. It leverages the spatial feature extraction capabilities of CNNs and the ability of MLPs to learn

complex nonlinear relationships. In fault classification, CNN-MLP models can effectively capture both spatial and non-spatial patterns in the sensor data. The CNN component of the model extracts spatial features, while the MLP component processes these features and performs the final classification.

As discussed in chapter two, there are two approaches to multi-class classification. Direct or native multi-class classification and binary transformation (class binarization) multi-class classification. In this study, due to the nature of the data, we used the binary transformation, where we trained 4 classifiers for each of the 4 fault classes, and during classification the classifier with the highest probability is chosen. Accordingly, the accuracy, precision, recall and f1 of each classifier has been shown. The confusion matrix and learning curve of each classifier has also been shown.

3.5. Design and Development Tools

3.5.1. Design Tools

Design tools have been used in this thesis to build and design the concepts of the proposed models. Two tools in particular, EdrawMax and Canva, have been used to design the proposed solution's various diagrams, flowcharts and the entire architecture.

EdrawMax[13]: EdrawMax is a diagramming software that allows users to create a wide range of diagrams, including flowcharts, mind maps, organizational charts, and network diagrams. It is designed for professionals and businesses that need to create detailed and technical diagrams. EdrawMax offers a vast library of templates and symbols, and its interface is customizable for specific needs

Canva[14]: Canva is a graphic design software that focuses on creating visually appealing designs for social media, marketing, and other purposes. Canva offers an extensive library of templates, images, fonts, and design elements to help users create professional looking designs without extensive design experience. It is suitable for individuals, small businesses, and marketers who want to create eye-catching designs quickly and easily.

[13] https://www.edrawsoft.com/ad/edrawmax/
[14] https://www.canva.com/

3.5.2. Development Tools

This includes the hardware and software tools and other platforms that were used to create, test and implement the proposed solution.

Hardwares: The hardware tools that were involved in implementing the proposed solution and carrying out the whole research are:

- RAM: Random Access Memory is a type of hardware used in computers and other digital devices to temporarily store data that the device needs to access quickly.
- Hard disk: Hard disks are non-volatile hardwares that provide a large and relatively inexpensive form of storage that can hold vast amount of data. They are used to store wide variety of digital data including the OS, software applications, video, and music.
- GPU: Graphics Processing Unit is a specialized hardware designed specifically for handling large amounts of parallel mathematical computations. Over time more and more developers are turning to GPUs to accelerate computations in fields such as machine learning, scientific simulation and others.

Softwares: The software tools involved in implementing the proposed solution i.e., writing and testing the code, training the models and others include:

Visual Studio Code[15]: VS Code is a lightweight yet powerful editor for coding, debugging, and building software applications. It offers a wide range of features and extensions, making it a popular choice among developers. It has built-in debugger, support for Git version control, and integrations with many popular programming languages. Thanks to its python and Jupyter extensions and features such syntax highlighting, code completion and debugging, it has proved to be a great choice for machine learning tasks as well.

Python[16]: Python is a popular programming language for machine learning due to its simplicity, ease of use, and wide range of libraries and frameworks that are available for machine learning development. In fact, due to the availability of libraries such as NumPy, Pandas, and Matplotlib, which provide powerful tools for data manipulation, analysis and visualization that makes it easy to work on large datasets and perform complex operations on data, it has become the de facto language for many machine learning developers.

[15] https://code.visualstudio.com/
[16] https://www.python.org/

Keras[17]: Keras is a high-level neural network API. It provides a user-friendly interface for building neural networks, allowing developers to create complex models with just a few lines of code. It was developed to enable fast experimentation with deep neural networks, allowing researchers and developers to quickly prototype and test different models and architectures.

TensorFlow[18]: TensorFlow is an open-source machine learning framework designed to simplify the process of building and training machine learning models, particularly for large-scale, complex datasets. It provides a wide range of tools and APIs for building and training the models, including low-level API for custom building and high-level APIs such as Keras.

Google Colab[19]: Google Collaboratory is a free, cloud-based platform for developing and running machine learning models. The main benefit of google colab is it provides access to powerful hardware, including GPUs and TPUs, which can significantly accelerate the training process.

GitHub[20]: GitHub is a web-based platform for version control and collaboration that is widely used in software development. We used GitHub as a means of backup and also track changes to the code. It also allowed us to work from multiple locations and computers by just pulling the latest update from the repo and then continue working on it, then we push the changes.

On the document writing side we used:

Office[21]: Office is a suite of software applications developed by Microsoft corp. We used a variety of office applications such as Word, PowerPoint and Excel for document writing, presentations and data cleaning and visualization.

Mendeley cite[22]: Mendeley cite is a citation tool developed by Mendeley designed to help researchers and academics manage and cite their references in their academic papers and researches. It can be integrated with Microsoft Word to make it easier to add references to documents.

[17] https://keras.io/
[18] https://www.tensorflow.org/
[19] https://colab.research.google.com/
[20] https://github.com/
[21] https://www.microsoft.com/en-us/download/office.aspx
[22] https://www.mendeley.com/reference-management/mendeley-cite

3.6. Performance Evaluation Metrics

Performance evaluation is an essential aspect of model development. It refers to the process of assessing the accuracy and efficiency of a model's predictions on a given dataset. The goal is to determine how well a model is performing in solving a specific problem. The information can be used to improve the model's accuracy and efficiency, as well as to compare different models and select the best one for the given task. There are many different metrics that can be used but of course it is important to choose the right metric for a given task, i.e., some metrics should be used for regression problems while others for classification.

3.6.1. Forecasting Analysis Evaluation

In forecasting, the goal is to predict a continuous variable or a time-series value by analyzing the past values of the time-series. Some of the commonly used performance evaluation metrics in forecasting are:

1. **Mean Absolute Error (MAE)**

Mean Absolute Error measures the average absolute difference between the predicted and actual value. The absolute is used to make the number positive even if the difference between the predicted and actual values becomes negative. The equation for MAE is(Schneider & Xhafa, 2022):

$$MAE = \frac{1}{N}\left(\sum_{i=1}^{N} |(Actual - predicted\ value)|\right)$$

Equation 3-1. Mean Absolute Error (MAE)

2. **Mean Square Error (MSE)**

Mean Squared Error measures the average of the squared differences between the predicted and actual values. The differences are squared in order to eliminate negative values and to penalize large errors than smaller errors[23]. The formula is given as:

$$MSE = \frac{\sum_{i=1}^{N} (Actual(i) - predicted(i))^2}{N}$$

Equation 3-2. Mean Square Error (MSE)

3. **Root Mean Squared Error (RMSE)**

[23] https://statisticsbyjim.com/regression/mean-squared-error-mse/

This is similar to MSE, but it takes the square root of the MSE to provide a more interpretable value. It measures the average of the square of the difference between actual and predicted values (Bouktif et al., 2018). The formula is given as:

$$RMSE = \sqrt{\sum_{i=1}^{N} \frac{(predicted(i) - Actual(i))2}{N}}$$

Equation 3-3. Root Mean Squared Error (RMSE)

4. **Mean Absolute Percentage Error (MAPE)**

The mean absolute percentage error measures the average percentage difference between predicted and actual values. MAPE can be used to measure the prediction accuracy for a given forecasting model (Kim & Kim, 2016). The formula is given as:

$$\text{MAPE} = \frac{100}{N} \sum_{t=1}^{N} \left| \frac{Actual - predicted}{Actual} \right|$$

Equation 3-4. Mean Absolute Percentage Error (MAPE)

3.6.2. Classification Analysis Evaluation

In classification the goal is to predict the class or category of a given instance based on its features or attributes. Several metrics are used to evaluate the performance of a classification model although the choice of metric depends on the specific problem.

1. **Accuracy**

Is the most commonly used metric in classification analysis. It measures the proportion of correct predictions made by the model over the total number of predictions. In other words, it is a measure of how often the model correctly predicts the presence or absence of faults in the data.

$$Accuracy = \frac{TP + TN}{TP + FN + TN + FP\ (total\ instances)}$$

Equation 3-5. Accuracy formula

2. **Precision**

Precision measures the proportion of true positives among all the positive predictions made by the model. In other words, precision measures how often the model correctly predicts a fault when there is one.

$$Precision = \frac{TP}{TP + FP}$$

Equation 3-6. Precision formula

3. **Recall**

Recall measures the proportion of true positives among all the actual positive instances in the dataset. In other words, recall measures how many of the actual faulty data are correctly predicted as faulty by the model.

$$Recall = \frac{TP}{TP + FN}$$

Equation 3-7. Recall formula

4. **F1 score**

F1 score is the harmonic mean of precision and recall, and provides a balanced evaluation of the model's performance on both metrics. In our context, the F1 score would provide a single measure of the model's overall performance on predicting the presence or absence of faults in the sensor data, considering both false positives and false negatives.

$$F1\ score = 2 * \frac{Recall * Precision}{Recall + Precision}$$

Equation 3-8. F1 score formula

5. **Confusion matrix**

This is a tabular representation of the performance of the model which shows the number of correct and incorrect predictions made by the model on the dataset. The matrix consists of four values: true positives (TP), false positives (FP), false negatives (FN) and true negatives (TN).

	Predicted Normal	Predicted Bias	Predicted Drift	Predicted Polydrift	Predicted Random
Actual Normal	TN	FP	FP	FP	FP
Actual Bias	FP	TN	FP	FP	FP
Actual Drift	FP	FP	TN	FP	FP
Actual Polydrift	FP	FP	FP	TN	FP
Actual Random	FP	FP	FP	FP	TN

Table 3-2. Confusion matrix table

CHAPTER FOUR

PROPOSED SOLUTION ARCHITECTURE

4.1. Introduction

So far, we have discussed the literature review, the research methodology, the gap we found in the literature, i.e., the problem statement and raised some research questions. In this section we will describe in detail the architecture of the solution we proposed to fill in the gap mentioned and answer the research questions raised. We will also discuss the procedures and steps followed including algorithm selection. Overall, this chapter aims to describe the architecture used for the sensor fault prediction and classification.

4.2. Proposed Sensor Fault Prediction and Classification Architecture

One of the major gaps we concluded from the literature review was that most of the works on sensor faults are focused on detecting faults. While detecting faults is useful, it is however a reactive method, i.e., it addresses the problem (detects faults) after it has occurred. Our proposed system predicts potential faults or anomalies before they occur based on the sensor's historical data.

The proposed architecture has 2 models that work in two phases. In the first phase, a forecasting model that is trained on the historical measurements of the sensors is used to forecast the *N* future values of each sensor. In the second phase, a classification model that is trained on the fault injected dataset, is used to classify the forecasted values. The fault injected dataset has 4 classes for the 4 fault types and a 5^{th} class to represent the normal (clean) values. So, in the second phase, a multi class (5 classes, i.e., normal, bias, drift, Polydrift and random) classification task is performed to classify the forecasted values.

In the first phase, a univariate timeseries forecasting is performed. After loading all the sensors datasets, data integration is performed to combine the data from different sensors into 1 data frame. Then other preprocessing tasks such as resampling, data type conversion, data transformation, scaling and others are performed. Feature engineering tasks such as generating new time-based and rolling window statistics is performed. Finally, the data is converted to a supervised learning problem using sliding window. A regression model is then trained using CNN-LSTM with attention mechanism on the preprocessed data to forecast future *N* values

for all the sensors. The forecasted values are then passed to a CNN-MLP classifier which is trained on the fault injected data to give the classes of the values. The classes are normal, bias, drift, random or Polydrift.

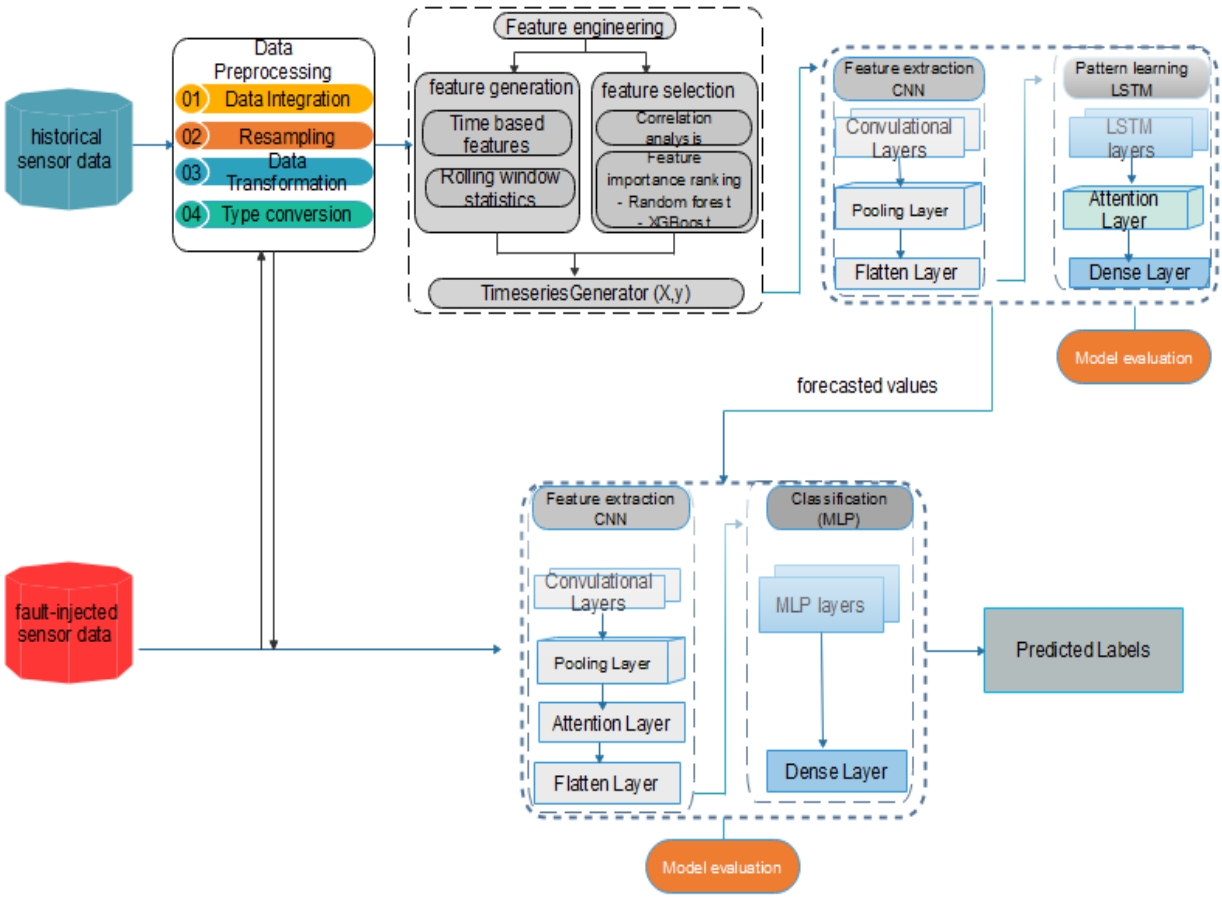


Figure 4-1. The proposed architecture for fault prediction and classification

Figure 4-1 shows the architecture of the proposed system. The architecture consists of a 2-step process, in which, in the first stage a CNN-LSTM model is trained on the historical sensor data. This model is then used to forecast future sensor measurements based on the historical data. In the second stage the forecasted measurements are then passed to the second model, CNN-MLP, which has been trained on fault-injected data to identify 4 types of faults, to categorize the measurements accordingly.

4.3. Data preprocessing phase

Data preprocessing is a very crucial step in data analysis. It is the process of cleaning, transforming, and organizing raw data before further analysis. It helps to ensure that the data is

accurate, consistent, and complete. Generally, it involves a series of steps that are designed to improve the quality and reliability of the data.

Sensor data are timeseries data. Timeseries data refers to a type of data that is collected over time at regular intervals. What makes this type of data unique is the temporal dependence between the observations, i.e., observations are not independent and are influenced by previous observations in the series. Characteristics such as seasonality, trends, patterns and irregularities need to be taken into consideration when modelling timeseries data.

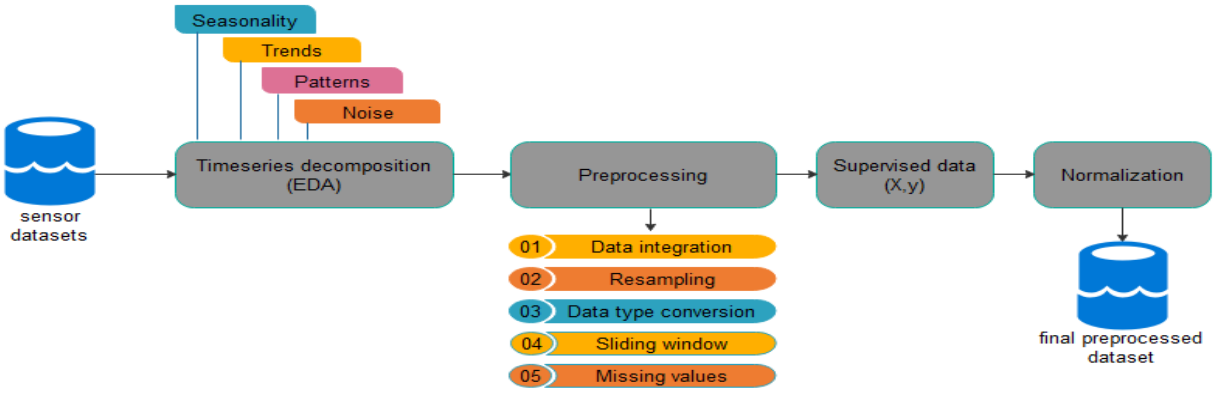


Figure 4-2. Data preprocessing for univariate timeseries forecasting

Figure 4-2 show the data preprocessing done for the first stage, i.e., forecasting. After performing timeseries decomposition and detrending the data by performing the differencing operation, preprocessing is then done on the data, i.e., integrating the data from the sensors into 1 dataframe, resampling the frequency to 2 minutes, converting the timestamp feature to datetime object and so on. Finally, we convert the timeseries data into a supervised learning problem by using the keras TimeSeriesGenerator API.

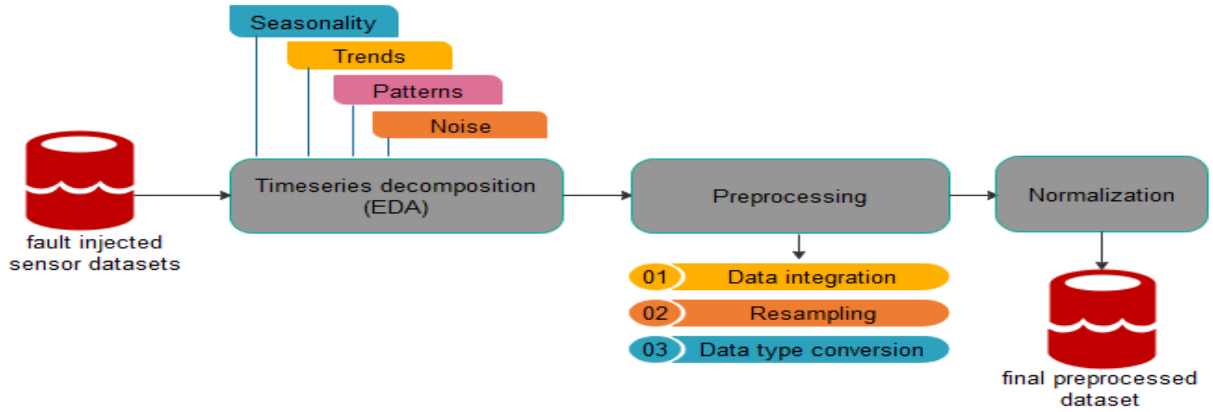


Figure 4-3. Data preprocessing for multi-class classification

Figure 4-3 show the similar preprocessing done for the second stage i.e., classification. The main difference being in this case the problem is already a supervised problem as we have inputs and output, where the output is the has_fault_type feature.

Timeseries Decomposition: Timeseries decomposition is a technique used in timeseries analysis to break down the time series data into its constituent components. Breaking a timeseries data into its components, which are usually the trend, seasonality and noise, can help us understand the underlying patterns and trends in the data in order to develop better predictive models.

There are many different methods for decomposing a timeseries data, but the most common approaches are additive and multiplicative approach. In the additive model, the different components are added together, and the overall time series is expressed as Y(t) = Trend(t) + Seasonality(t) + Noise(t). In this model, the size of the seasonal fluctuations remains constant over time, regardless of the magnitude of the trend or the level of the time series. This model is appropriate when the seasonal variations in the data are relatively stable and do not depend on the overall level of the time series. In the multiplicative model, the different components are multiplied together, and the overall time series is expressed as: Y(t) = Trend(t) * Seasonality(t) * Noise(t). In this model, the size of the seasonal fluctuations varies in proportion to the overall level of the timeseries. This model is appropriate when the fluctuations in the data are proportional to the overall level of the timeseries, and the size of the seasonal variations changes as the level of the time series changes.

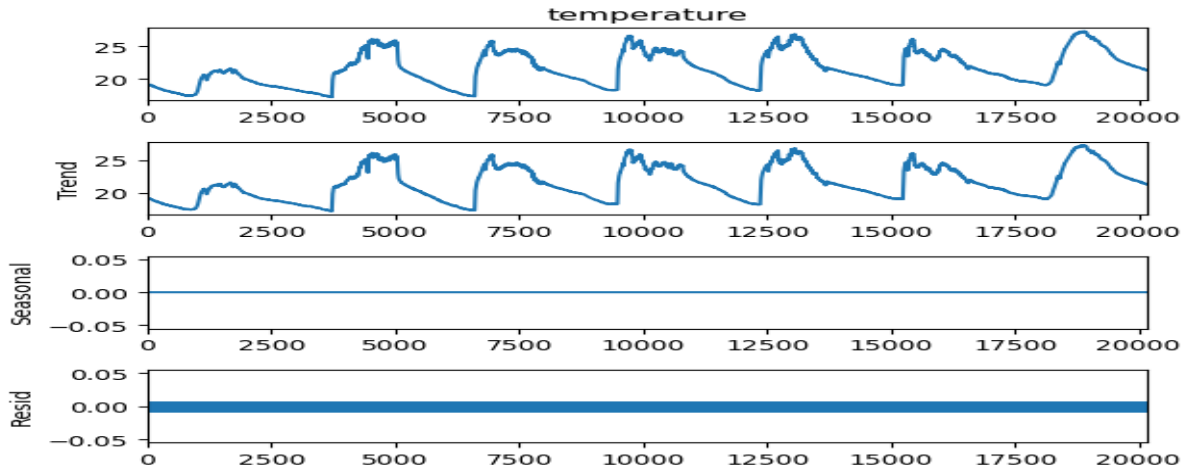


Figure 4-4. Node 1 temperature data decomposition

Figure 4-4 shows the timeseries decomposition for sensor 1 using the additive model. As can be observed the measurement does not contain any seasonality, however, there is an increasing trend component. To confirm we performed the ordinary least square (OLS) linear regression and the result indicated there is an increasing trend, so we performed a differencing operation to remove the trend.

Trend analysis: The trend component of a timeseries represents the long-term behavior or direction of the data over time. This component can help us analyze the data and determine whether the data is increasing, decreasing, or stationary.

Seasonality analysis: The seasonality component of a time series represents the periodic fluctuations in the data that occur at regular intervals, such as hourly, daily, weekly, monthly, or yearly cycles. By decomposing the timeseries data into seasonal and non-seasonal components, we can identify the time periods that are likely to exhibit seasonal patterns.

Noise analysis: The noise component of a timeseries represents the random fluctuations or errors in the data that cannot be explained by trend or seasonality. This analysis can help in identifying the sources of noise in the data and develop strategies for reducing or eliminating it.

4.4. Hybrid deep learning

Hybrid deep learning models are a combination of two or more deep learning models or architectures that are designed with the aim of improving the performance of a machine learning model. These models can be more effective than using a single deep learning technique alone, as they can leverage the strengths of the combined models to achieve better performance.

4.4.1. CNN-LSTM

A hybrid CNN-LSTM model is a deep learning architecture that combines convolutional neural networks (CNNs) and long short-term memory (LSTM) networks. The architecture is suitable for processing both spatial and temporal data, making it particularly effective for time-series forecasting problems. The model basically works in 3 main steps.

1. The input time series data is first fed into a CNN layer that extracts the important features from the data. The output of this layer is a set of high-level feature maps.

2. The output of the CNN layer is then fed into an LSTM layer that learns the temporal dependencies in the data. It takes the feature maps as input and produces a sequence of output vectors.
3. The output vectors of the LSTM layer are then fed into a fully connected layer that generates the final forecasts.

The architecture allows the model to learn both the spatial and temporal dependencies in the time series data. The CNN layer extracts the spatial features from the input data, while the LSTM layer captures the temporal dependencies in the data. The model is trained using the mean squared error (MSE) loss function. During training, the model learns to minimize the MSE between the predicted and actual values. To further improved the model, attention layer is added to the model and hyperparameter tuning is performed.

Hybrid CNN-LSTM Algorithm

Model building

Input: *preprocessed non-faulty sensor data*

Output: *A dictionary object containing forecasted values of all sensors identified by sensor_id*

_Begin()

Create a sequential model.

Add a time-distributed convolutional layer with 64 filters, a kernel size of 2, and ReLU activation function. Set the input shape to (n_timesteps, n_features, 1).

Add another time-distributed convolutional layer with 64 filters, a kernel size of 2, and ReLU activation function.

Add a time-distributed max pooling layer with a pool size of 2.

Add a time-distributed flatten layer.

Create a sequential model.

Add an LSTM layer with 32 units, a ReLU activation function, and return sequences set to True.

Add another LSTM layer with 32 units, a ReLU activation function, and return sequences set to True.

Add an attention layer (custom implementation).

Add a dropout layer with a rate of 0.4.

Add a dense layer with 1 unit.

Combine the CNN and LSTM models:

Create a sequential model.

Add the CNN model.

Add the LSTM model.

Compile the combined model:

Set the loss function to mean squared error (mse).

Set the optimizer to Adam with a learning rate of 0.001.

Train the model:

Fit the model using the generator for training data and the validation generator for validation data.

Set the number of epochs to 200.

Include the early stopping callback (es).

Set the verbose level to 1 to display progress during training.

Initialize sensor_predictionsCnnLstmAtt as an empty dictionary

For each unique sensor_id in data['mote_id']:

Get sensor_data by selecting rows where data['mote_id'] equals sensor_id, and dropping the first column

Transform the sensor_data using the MinMaxScaler object

Get the predictions by calling the make_predictions() function with cnnLstmAtt and sensor_data as input

Reshape the predictions as a 1D array

Add the predictions to the sensor_predictionsCnnLstmAtt dictionary with sensor_id as key

Return: sensor_predictionsCnnLstmAtt{}

End

Multi-step forecasting function

Input: *trained model, last n_timesteps values of a sensor*

Output: *n_future steps forecasted values for the sensor*

Begin:

make_predictions(model, sensor_data):

Initialize an empty list called predictinos

for I in range (number of future timesteps)

Get the last n_timesteps rows of scaled_train as first_eval_batch

Reshape first_eval_batch as a 3D array with shape (1, n_timesteps, n_features)

Get the predictions for the current batch and select the first element of the resulting array

Create a dataframe df with temperature, mean, variance, and std from the scaled_train data

Select only the temperature column of df

Reshape the current_pred array as a 1D series using pd.series()

Append the current_pred series to the temperature column of df and remove the first row

Set scaled_train to the new temperature column of df and add new columns for mean, variance, and std using rolling window function with window_size as defined in window_size

Fill any NaN values in scaled_train with mean of non-NaN values

Inverse transform the scaled_train data

Get the last value of the first column of the inversed data as pred1

Append pred1 to the predictions list

Convert scaled_train to a numpy array

Convert predictions to a numpy array

Return predictions

End

4.4.2. CNN-MLP

A hybrid CNN-MLP model is a deep learning architecture that combines convolutional neural network (CNNS) with multilayer perceptron (MLP). In the architecture, first the input data is fed into a CNN layer that extracts important features from the data. The output of this layer is a set of high-level feature maps. The output of the CNN layer is then flattened and fed into an MLP layer that produces a set of output probabilities for each class. The class with the highest probability output by the MLP is then determined to be the predicted class.

The architecture allows the model to learn both the spatial and semantic features in the input data. Since we are implementing binary transformation strategy for multi-class classification, four instances of the model are trained on the four sets of data and the instance that produces the highest probability determines the class. The instances (classifiers) are trained using binary_crossentropy as the loss function.

Hybrid CNN-MLP algorithm

Model building

Define the CNN model:

- Create a sequential model.
- Add a 1D convolutional layer with 32 filters, a kernel size of 2, ReLU activation function, and input shape (X_train_bias.shape[1], 1). Use padding='same' to maintain the input shape.
- Add a max pooling layer with a pool size of 1.
- Add another 1D convolutional layer with 16 filters, a kernel size of 2, and ReLU activation function. Use padding='same'.
- Add another max pooling layer with a pool size of 1.
- Add a flatten layer.

Define the MLP model:

- Create a sequential model.
- Add a dense layer with 64 units and ReLU activation function.
- Add a dropout layer with a rate of 0.2.
- Add another dense layer with 32 units and ReLU activation function.
- Add a dense layer with 1 unit and sigmoid activation function.

Combine the CNN and MLP models:

- Create a sequential model.
- Add the CNN model.
- Add the MLP model.

Compile the combined model:

- Set the loss function to binary cross-entropy.
- Set the optimizer to Adam with a learning rate of 1e-3.

- Use 'accuracy' as the evaluation metric.

Train the model:

- Fit the model using X_train_bias and y_train_bias as the training data, with a validation split of 0.2.
- Include the early stopping callback (es).
- Set the number of epochs to 500.
- Set the batch size to 50.

4.4. Hyper-parameter optimization

Hyperparameter optimization is the process of finding the best set of parameters for a machine learning algorithm to achieve optimal performance on a given task. Hyperparameters are parameters that are not learned during training, but rather are set before training begins. For example, learning rate, batch-size, number of epochs, number of hidden layers and others.

Grid search: Grid search is a brute force approach that involves evaluating the model performance for every possible combination of hyper parameters with in a defined search range that is manually given. It is exhaustive and computationally expensive for large search ranges.

Random search: This approach randomly samples hyperparameters from a given range and selects the combination that results in the best performance. It is often faster than grid search and can be more effective for high dimensional search ranges.

Bayesian optimization: This approach uses the bayes theorem to model the relationship between the hyperparameters and the objective function. This model is then used to guide the search for the best set of hyperparameters. This approach can be more efficient than grid and random search for complex search ranges.

Gradient-based optimization: This approach uses gradient descent and other optimization algorithms to minimize the objective function with respect to the hyperparameters. This approach can be more effective for deep neural networks and other complex models.

Cross-validation, regularization and early stopping are common techniques used when performing hyperparameter optimization to ensure the model generalizes well to unseen data. common

CHAPTER FIVE

IMPLEMENTATION OF THE PROPOSED SOLUTION

5.1. Overview

In this chapter we discuss the experiment and implementation of the proposed architecture for fault prediction and classification.

5.2. Environmental set-up

It is essential to have a working implementation environment for any research or system development to ensure that the proposed system can be properly trained, evaluated and tested. Various set of tools, hardware, software, and other resources have been used to develop, run and test the proposed system. We have divided them under two categories and discussed them below.

5.2.1. Tools and Packages

Deep learning model developments require the use of various tools and packages to enable efficient and effective implementation of models. These tools and packages provide essential and necessary functionalities for data preprocessing, model construction, training, and evaluation.

Tools and packages	***Version used***	***Brief description***
Python		*Popular programming used for DL due to its simplicity, versatility, and its richness in abundant libraries*
TensorFlow		*DL framework that provides efficient implementation of various neural network architectures*
Keras		*High level DL library with user friendly API for building neural networks.*
Scikit-Learn		*Python library that provides implementation of many ML algorithms and neural network models.*
Pandas		*Python library for data manipulation and analysis*

NumPy	*Python library for numerical computing that provides efficient array operations.*
Matplotlib	*Python library for data visualization and model outputs.*
Jupyter Notebook	*Web-based deep learning model development, experimentation, and sharing application. It also comes as a VSCode plugin.*

Table 5-1. Tools and packages used

5.2.3. Deployment Environment

The hardware and software resources used during the implementation of the proposed system are:

Desktop	Operating system*: windows 10* System*: 64-bit operating, x64-based processor* Processor*: Intel(R) Core (TM) i5-4590 CPU @ 3.30GHz 3.30 GHz* Memory *size: 8 GB*
Laptop	Model*: HP EliteBook 8470p* Operating system*: windows 10* Memory size*: 4 GB* System type*: 64-bit operating system, x64-based processor*

Table 5-2. Hardware resources

5.3. Fault prediction and classification implementation

5.3.1. Data preprocessing implementation

Data preprocessing is a crucial step in data analysis that involves transforming and preparing raw data into a format that is suitable for analysis. It ensures that the data is clean, consistent, and usable.

preprocessing libraries

```
#import libraries
import pandas as pd
from sklearn.preprocessing import MinMaxScaler
from keras.preprocessing.sequence import TimeseriesGenerator
```

loading data into the environment

```
#Read in the sensor datasets
node1Temp= pd.read_csv('clean/Node01Temp.csv',delimiter=',',
index_col ='timestamp', parse_dates=True)
node2Temp= pd.read_csv('clean/Node04Temp.csv',delimiter=',',
index_col='timestamp', parse_dates=True)
node3Temp= pd.read_csv('clean/Node06Temp.csv',delimiter=',',
index_col='timestamp', parse_dates=True)
node4Temp= pd.read_csv('clean/Node19Temp.csv',delimiter=',',
index_col='timestamp', parse_dates=True)
node5Temp= pd.read_csv('clean/Node22Temp.csv',delimiter=',',
index_col='timestamp', parse_dates=True)
node6Temp= pd.read_csv('clean/Node23Temp.csv',delimiter=',',
index_col='timestamp', parse_dates=True)
node7Temp= pd.read_csv('clean/Node29Temp.csv',delimiter=',',
index_col='timestamp', parse_dates=True)
node8Temp= pd.read_csv('clean/Node33Temp.csv',delimiter=',',
index_col='timestamp', parse_dates=True)
node9Temp= pd.read_csv('clean/Node34Temp.csv',delimiter=',',
index_col='timestamp', parse_dates=True)
node10Temp= pd.read_csv('clean/Node45Temp.csv',delimiter=',',
index_col='timestamp', parse_dates=True)
```

data integration

```
# Concatenate the training datasets into a single dataframe
data = pd.concat([node1train,node2train,node3train,node4train,
node5train,node6train,node7train,node8train,node9train,node10train,],
ignore_index=True)
```

Data scaling

```
# Scale the training data using MinMaxScaler
```

```
scaler = MinMaxScaler()
scaler.fit(data)
scaled_train = scaler.transform(data)
```

converting to supervised learning problem

```
# Define the timesteps(lag), future timesteps and no. of features
n_timesteps = 3 # number of timesteps to use for each sample
out_future = 1  # number of future timesteps to predict
n_features = 4  # number of features for each sensor
# Using the TimeSeriesGenerator create input output sequence
(pair) for supervised learning
generator= TimeseriesGenerator(scaled_train[:,1:], scaled_train[:,1],
length=n_timesteps, batch_size=200)
```

Balancing fault injected dataset

Highly imbalanced dataset can lead to biased model during training which can result in poor predictive performance especially on the minority classes. Out dataset had a significantly higher number for normal class compared to others and class 1 had the least. We experimented with under sampling, over sampling and SMOTE to find out which works better for our dataset.

Under-sampling

```
#under sampling
from imblearn.under_sampling import RandomUnderSampler
rus = RandomUnderSampler(random_state=0) # Numerical value
X_res, y_res = rus.fit_resample(X, y)
ax = y_res.value_counts().plot.pie(autopct='%.2f')
```

Over-sampling

```
#over sampling
from imblearn.over_sampling import RandomOverSampler
rus = RandomOverSampler(random_state=0) # Numerical value
# rus = RandomUnderSampler(sampling_strategy="not minority") # String
X_sampled, y_sampled = rus.fit_resample(X, y)
ax = y_sampled.value_counts().plot.pie(autopct='%.2f')
```

Smote sampling

```
#smote imbalance handling
from imblearn.over_sampling import SMOTE
smote= SMOTE()
X_smote, y_smote= smote.fit_resample(X, y)
ax = y_smote.value_counts().plot.pie(autopct='%.2f')
```

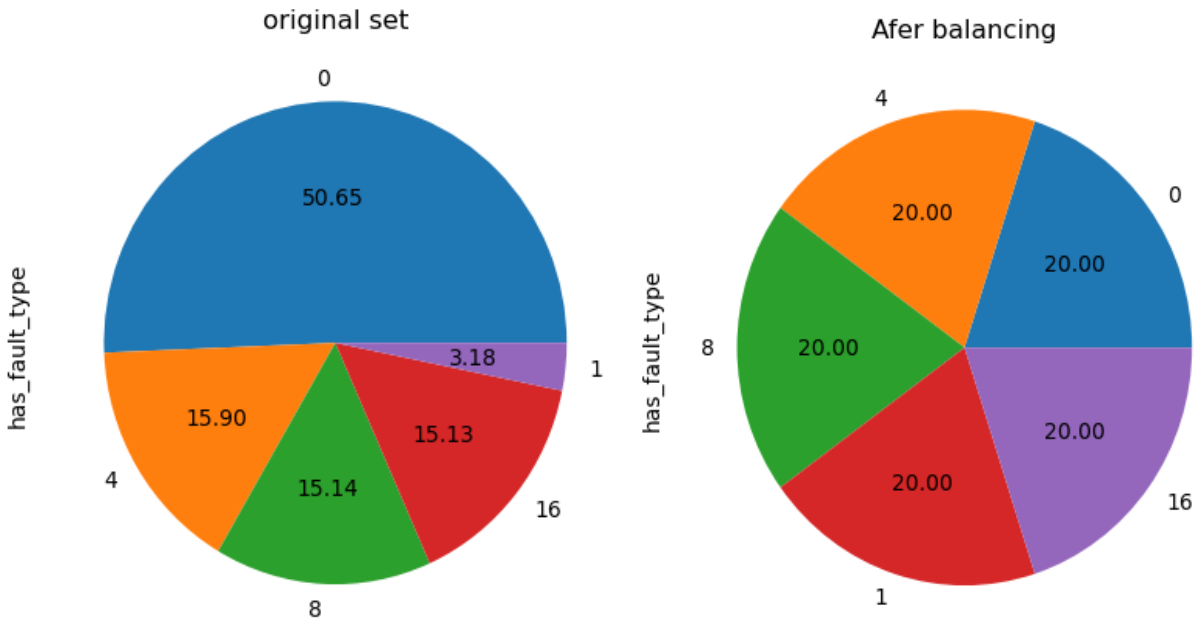


Figure 5-1. Data balancing

Figure 5-1 shows the result of data balancing. On the left we can see the original set with each class in terms of their percentage. It can clearly be seen there is an imbalance between the classes. On the right side we see the classes have been balanced.

5.3.2. Model implementation

Model implementation is a crucial step in data-driven research such as ours. In this section, we'll discuss the tools and algorithms and the optimization techniques used. We'll discuss the step-by-step procedure involved, starting from importing the necessary libraries, to creating the instances of the models to training and evaluating them.

Importing libraries

```
#@title importing libraries
```

```
from tensorflow import keras
from keras.models import Sequential
from keras.layers import LSTM, Dense, Dropout, Bidirectional, GRU, Flatten,
TimeDistributed, ConvLSTM2D
from keras.layers.convolutional import Conv1D, MaxPooling1D
from attention.attention import Attention_class
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
from matplotlib import pyplot as plt
import seaborn as sns
from keras.layers import LeakyReLU
from sklearn.metrics import precision_score, recall_score, f1_score
from sklearn.model_selection import StratifiedKFold
from sklearn.preprocessing import LabelEncoder
from keras.utils import to_categorical
from keras.callbacks import EarlyStopping
```

5.3.2.1. Hyperparameter tuning

Hyperparameter optimization is the process of finding the best set of parameters for a machine learning algorithm to achieve optimal performance on a given task. We used the keras RandomizedSearchCV class with n_iter of 10. We cross validated using TimeSeriesSplit with n_splits of 2 and scoring of neg_mean_squared_error. EarlyStopping with patience of 10 was used to avoid overfitting.

CNN-LSTM with Attention Regression parameter tuning		
Parameter name	Search space	Optimal value
Filters	[16, 32, 64]	64
Kernel_size	[2, 3]	2
Pool_size	[2]	2
Lstm_units	[32, 64, 128]	32
Dropout_rate	[0.2, 0.4, 0.5]	0.4
Learning_rate	[0.001, 0.01, 0.1]	0.001
Activation	Relu	Relu

Optimizer	Adam	Adam

Table 5-3. Regression parameters

Table 5-3 shows the result of the hyperparameter tuning operation. RandomizedSearchCV with 10 combination was implemented, and the table show the result of the combination that gave the highest score. The first column shows the hyper-parameters tuned, the second column shows the search space for each hyperparameter and the third column shows the optimal value chose.

```
cnn_model = keras.models.Sequential([
    keras.layers.TimeDistributed(keras.layers.Conv1D(filters=64,kernel_size=2,
    activation='relu'), input_shape=(n_timesteps, n_features, 1)),
    keras.layers.TimeDistributed(keras.layers.Conv1D(filters=64, kernel_size=2,
    activation='relu')),
    keras.layers.TimeDistributed(keras.layers.MaxPooling1D(pool_size=2)),
    keras.layers.TimeDistributed(keras.layers.Flatten()),
])

lstm_model = keras.models.Sequential([
    keras.layers.LSTM(32, activation='relu', return_sequences=True),
    keras.layers.LSTM(16, activation='relu'),
    Attention_class(),
    keras.layers.Dropout(0.4),
    keras.layers.Dense(1),
])

cnnLstmAttention = keras.models.Sequential([
    cnn_model,
    lstm_model,
])

cnnLstmAttention.compile(loss='mse', optimizer=Adam(learning_rate=0.001))
```

Figure 5-2. CNN-LSTM with Attention layer implementation

Figure 5-2 shows the implementation of the CNN-LSTM model using the keras sequential API. As can be seen the two models are separately defined and then combined together. It is then compiled using the MSE as the loss function and Adam optimizer with the learning rate value obtained from the hyper-parameter tuning.

```
# Create a function to make predictions for the next n_future timesteps for each sensor by taking a specific model as argument

def make_predictions(model, scaled_train):

   predictions=[]

   for i in range(n_future_timesteps):

     first_eval_batch=scaled_train[-n_timesteps:]

     current_batch=first_eval_batch.reshape(1,n_timesteps, n_features)

     current_pred= model.predict(current_batch)[0]

     df = pd.DataFrame(scaled_train, columns = ['temperature','mean',min,'max'])

     df=df['temperature']

     current_pred=pd.Series(current_pred.reshape(-1))

     df=df.append(current_pred, ignore_index=True)

     df=df[1:,]

     scaled_train=df.to_frame(name='temperature')

     scaled_train['mean'] = scaled_train['temperature'].rolling(window=window_size).mean()

     scaled_train[min] = scaled_train['temperature'].rolling(window=window_size).min()

     scaled_train['max'] = scaled_train['temperature'].rolling(window=window_size).max()

     scaled_train.fillna(scaled_train.mean(numeric_only=True).round(1), inplace=True)

     inverse_train=sensor_scaler.inverse_transform(scaled_train)

     pred1=inverse_train[-1:,0]
```

```
    predictions.append(pred1)
    scaled_train=scaled_train.to_numpy()
  predictions=np.array(predictions)
  return predictions
```

Figure 5-3. Implementation of recursive multi-step forecasting

Figure 5-3 shows the implementation of the multi-step forecasting. As can be seen we used the recursive approach as opposed to the direct multi-step forecasting. The recursive approach takes each forecasted timestep as input for the next timestep. This allows the model to capture new events that may happen as it forecasts further into the future.

```
#define CNN model
cnn_model = keras.models.Sequential([
                    keras.layers.Conv1D(filters=32, kernel_size=2, activation='relu',
                            input_shape=(X_train.shape[1],1), padding='same'),
        keras.layers.MaxPooling1D(pool_size=1),
        keras.layers.Conv1D(filters=16, kernel_size=2, activation='relu', padding='same'),
        keras.layers.MaxPooling1D(pool_size=1),
        Attention_class(),
        keras.layers.Flatten(),
  ])

# Define the MLP model
mlp_model = keras.models.Sequential([
      keras.layers.Dense(32, activation='relu'),
      keras.layers.Dropout(0.2),
      keras.layers.Dense(16, activation='relu'),
      keras.layers.Dense(1, activation='sigmoid'),
  ])
```

```
#Combine the CNN and MLP models
CnnMlp = keras.models.Sequential([
    cnn_model,
    mlp_model,
   ])
# Compile the model
CnnMlp.compile(loss='binary_crossentropy', optimizer=Adam(learning_rate=0.001), metrics=['accuracy'])
```

Figure 5-4. CNN-MLP implementation

Figure 5-4 shows the implementation of the CNN-MLP model using the keras sequential API. As can be seen the two models are separately defined and then combined together. It is then compiled using the binary_crossentropy as the loss function because we are using the binary transformation approach to multi-class classification, and Adam optimizer with the learning rate value obtained from the hyper-parameter tuning.

5.3.2.2. Model evaluation implementation

For the CNN-LSTM model we used regression metrics such as RMSE, MSE and MAE to evaluate the performance of the model. Since the model performs forecasting for each sensor, initially, the performance metrics were calculated for each individual sensor and then the results were added together and divided by the number of sensors. The average RMSE, MSE and MAE was then taken as the final measurement.

```
# Evaluate using evaluation metrics i.e., rmse, mse, mae
from sklearn.metrics import mean_squared_error, mean_absolute_error
from math import sqrt

num_sensors= {1,4,6,19,22,23,29,33,34,45} #list containing the sensor id's
rmse=0
mse=0
mae=0
test_sensor_id=0
for s in (num_sensors):
```

```
    test_sensor_id=test_sensor_id+1
    rmse= rmse + sqrt(mean_squared_error(test[test_sensor_id], cnnLstmAtt[s]))
    mse= mse + mean_squared_error(test[test_sensor_id], cnnLstmAtt [s])
    mae= mae + mean_absolute_error(test[test_sensor_id], cnnLstmAtt [s])
rmse=rmse/len(num_sensors)
mse= mse/len(num_sensors)
mae=mae/len(num_sensors)
```

Figure 5-5. Implementation of CNN-LSTM forecasting evaluation

Figure 5-5 shows the implementation of the forecasting evaluation, where we evaluate the rmse, mse, and mae of all the sensors and then take the average of them.

for classification, we used classification metrics such as accuracy, precision, recall and f1 evaluated using k-fold cross-validation with both 5 and 10 folds.

```
# Perform stratified k-fold cross-validation for bias
from sklearn.model_selection import StratifiedKFold
n_folds = folds
skf = StratifiedKFold(n_splits=n_folds, shuffle=True)
accuracy_scores_bias = []
precision_scores_bias = []
recall_scores_bias = []
f1_scores_bias = []
for fold_idx, (train_index, test_index) in enumerate(skf.split(X_bias,y_bias)):
     print(f"Fold {fold_idx + 1}")
      .
      .
      .
      biasCnnMlp.fit(X_train_bias, y_train_bias, validation_split=0.2, callbacks=[es],
      epochs=100, batch_size=50, verbose=1)
      #make predictions on test set
      #bias test
      predictions_b = biasCnnMlp.predict(X_test_bias)
```

```
        predicted_classes_b = np.argmax(predictions_b, axis=1)

        #calculate the measurements
        accuracy = np.mean(predicted_classes_b == y_test_bias)
        precision= precision_score(y_test_bias, predicted_classes_b, average='weighted')
        recall= recall_score(y_test_bias, predicted_classes_b, average='weighted')
        f1= f1_score(y_test_bias, predicted_classes_b, average='weighted')

        #append the measurements of each fold
        accuracy_scores_bias.append(accuracy)
        precision_scores_bias.append(precision)
        recall_scores_bias.append(recall)
        f1_scores_bias.append(f1)

# Calculate the average accuracy, precision, recall, and F1-score across all the folds
avg_accuracy = np.mean(accuracy_scores_bias)
avg_precision = np.mean(precision_scores_bias)
avg_recall = np.mean(recall_scores_bias)
avg_f1 = np.mean(f1_scores_bias)
```

Figure 5-6. K-Fold cross validation implementation

Figure 5-6 shows the k-fold cross validation implementation for the classification task. The implementation was run twice with folds = 5, and folds=10, for each classifier.

CHAPTER SIX

RESULTS AND DISCUSSION

6.1. Introduction

In this chapter we will discuss and summarize the findings and experiment results we got. As mentioned before, our overall architecture consists of two parts. The first part consists of a regression task where we forecasted n_future_timesteps values for each sensor. Under this part we experimented with GRU, LSTM, BiLSTM, CNN-GRU, CNN-LSTM and CNN-BiLSTM. The second part consists of a classification task, where we classified the forecasted values as normal, bias, drift, Polydrift or random faults. Under this part we experimented with MLP, CNN and CNN-MLP. We also discuss the results of the different hyper-parameter combinations ran during the random search hyperparameter tuning.

6.2. Feature Importance and Selection Results

Although deep learning models are designed to automatically learn and extract relevant features from raw timeseries data, manual feature engineering such as generating statistical features, time-based features and rolling window statistics can help to further enhance the feature representation. Incorporating such additional information to the model could allow the model to capture additional information from the data that might not be explicitly captured from the raw values alone. Accordingly, we generated both time-based and rolling windows features and then performed feature selection to identify more relevant features.

6.2.1. Correlation analysis

The correlation plot below shows the correlation of each feature with the target feature i.e., temperature. Accordingly, we can see that rolling mean, rolling min and rolling max have the highest correlation with the temperature.

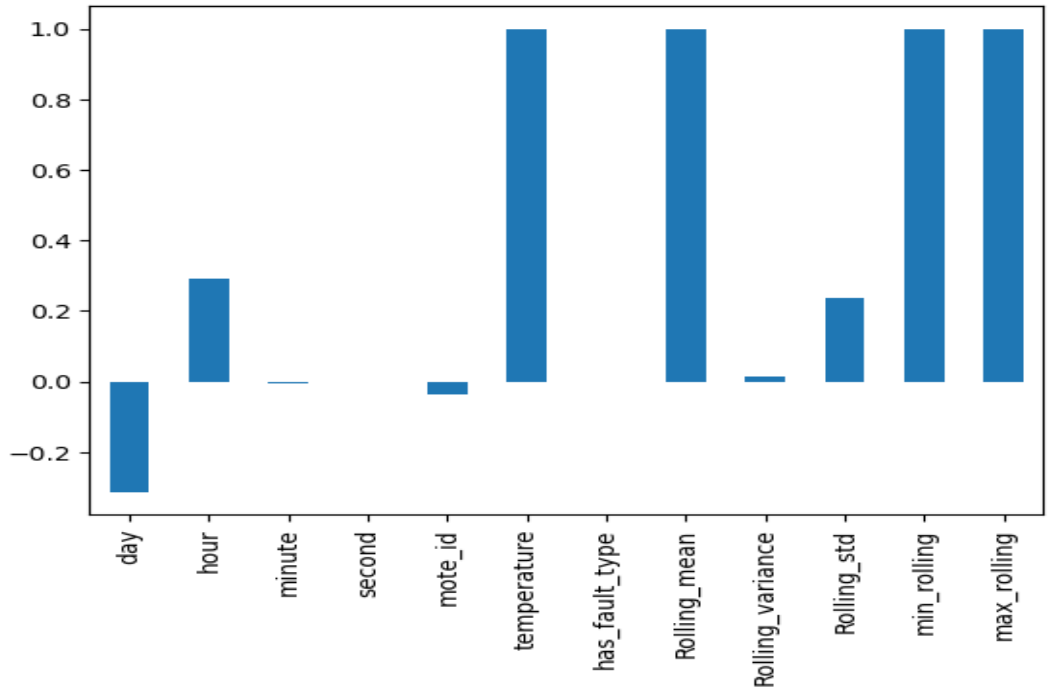


Figure 6-1. Correlation analysis result

Figure 6-1 shows the results of the correlation analysis of the different features generated against the temperature. Accordingly, it can be observed that rolling mean, rolling min and rolling max are highly correlated with the temperature features.

6.2.2. Feature importance ranking

Feature importance ranking is a way to assess the relevance or contribution of each feature. These techniques help to identify which features have the greatest impact on the model's prediction performance and can be used for feature selection. For this study we experimented with random forest regressor and XGBClassifier to calculate the importance of each feature.

6.4.2.1. Random Forest

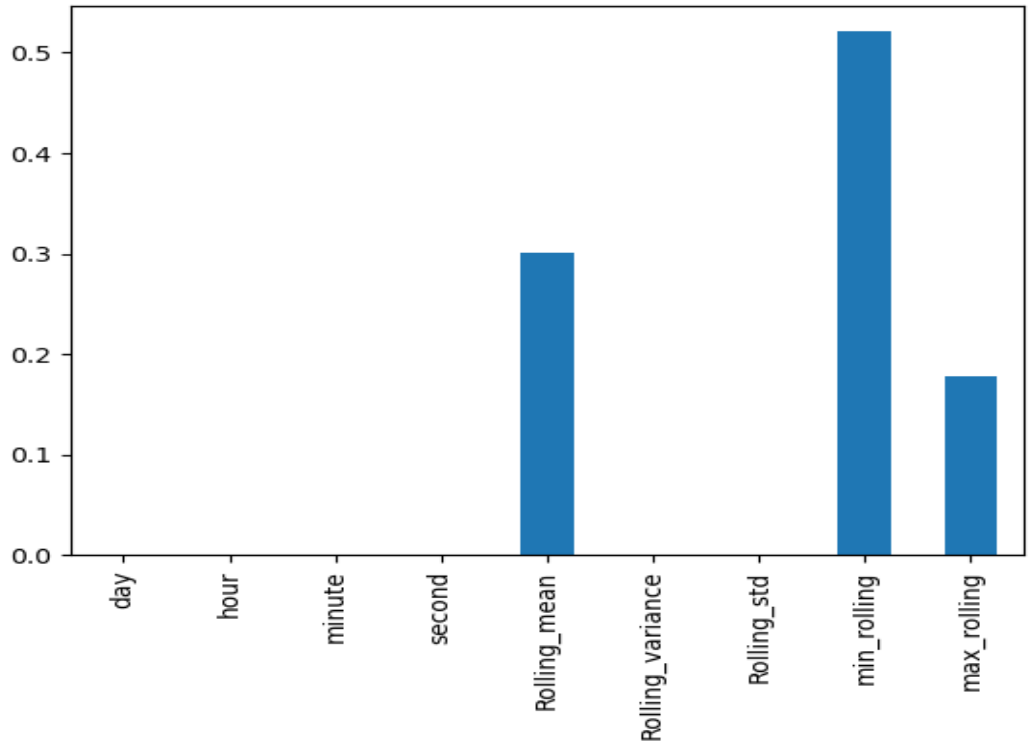


Figure 6-2. Random forest feature importance ranking result

Figure 6-2 shows the importance of each feature according to random forest regressor. RandomForestRegressor calculates feature importance based on the average impurity reduction or mean decrease in node impurity caused by splitting on a particular feature across all the decision trees in the forest. Features with higher average impurity reduction are considered more important. Accordingly, the rolling mean, rolling min and rolling max have been identified to be the most important features compared to the rest.

6.4.2.1. XGBClassifier

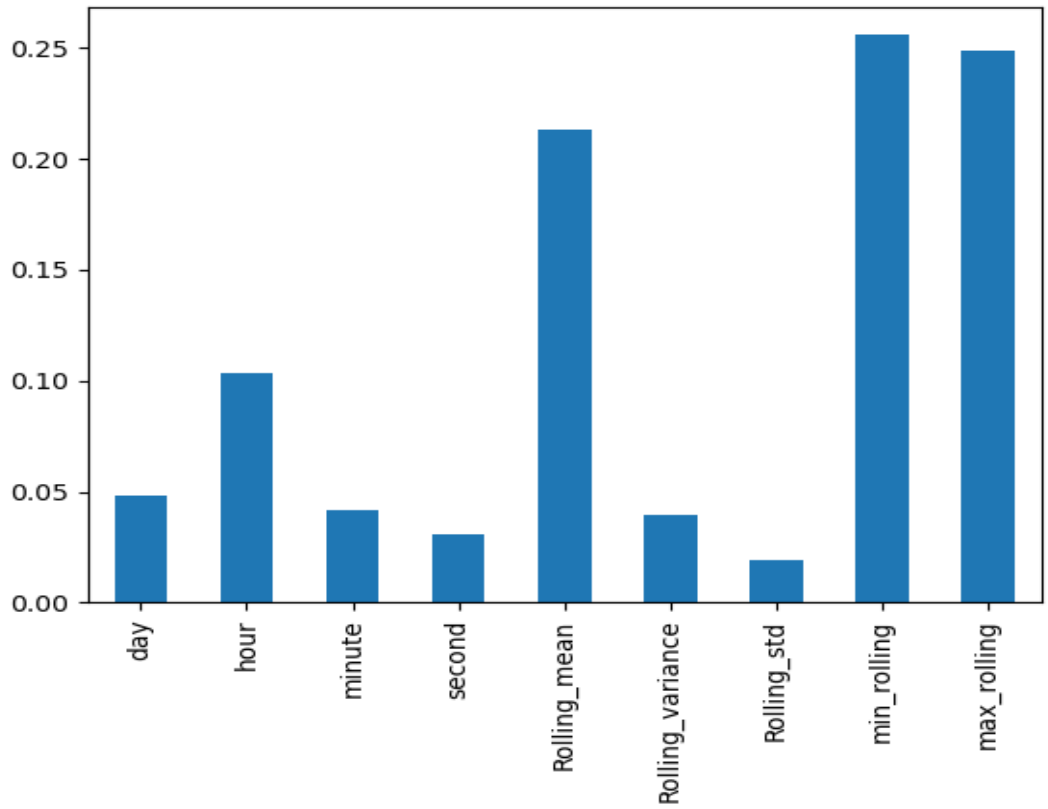


Figure 6-3. XGBClassifier feature importance ranking result

Figure 6-3 shows the results of XGBClassifier. XGBClassifier is an implementation of the gradient boosting algorithm. The classifier provides feature importance scores based on the number of times a feature is used to split the data across all the trees in the model. Features with higher occurrences of being used for splitting are considered more important. Accordingly, it can be seen that rolling mean, rolling min and rolling max are ranked highest.

6.3. Model Performance Results

6.3.1. Regression Analysis Results

6.3.1.1 TimeSeriesSplit(n_splits=2)

Traditional k-fold cross validation cannot be directly used when performing forecasting tasks using historical data because maintaining the temporal dependency between the data and making sure that future information leakage does not happen is very important in forecasting. The keras TimeSeriesSplit function extends the k-fold cross validation, and instead of random shuffling, splits the data into consecutive chronological folds.

Models	n_splits=2, n_future_timesteps=45		
	RMSE	MSE	MAE
GRU	3.6437	15.9823	3.6413
LSTM	2.7529	7.5784	2.7180
BiLSTM	2.6931	8.9624	2.5233
CNN-GRU	3.1501	11.2578	3.0121
CNN-LSTM	2.1581	6.4141	2.1403
CNN-BiLSTM	2.5565	8.1441	2.4237

Table 6-1. Forecasting evaluation before feature-engineering and selection

Table 6-1 shows the evaluation result gained when the models were trained just on the historical measurements of the sensors, with no additional features generated through feature engineering. Although the performances of the models are relatively close, the CNN-LSTM model outperformed the others scoring the least RMSE of 2.1581 for the 45 multisteps forecasting.

Models	n_splits=2, n_future_timesteps=45		
	RMSE	MSE	MAE
GRU	3.6353	13.2154	3.4260
LSTM	2.4569	8.4576	2.3862
BiLSTM	2.3558	7.0005	2.3324
CNN-GRU	2.2061	6.0298	2.1585
CNN-LSTM	2.1037	5.7972	2.0957
CNN-BiLSTM	2.2522	6.4196	2.2442

Table 6-2. Forecasting evaluation after feature-engineering and selection

Table 6-2 shows the performance of the models after feature engineering, i.e., generation of time-based and statistical rolling window features, which generally gives additional information about the distribution and central tendency of the data as input to the model.

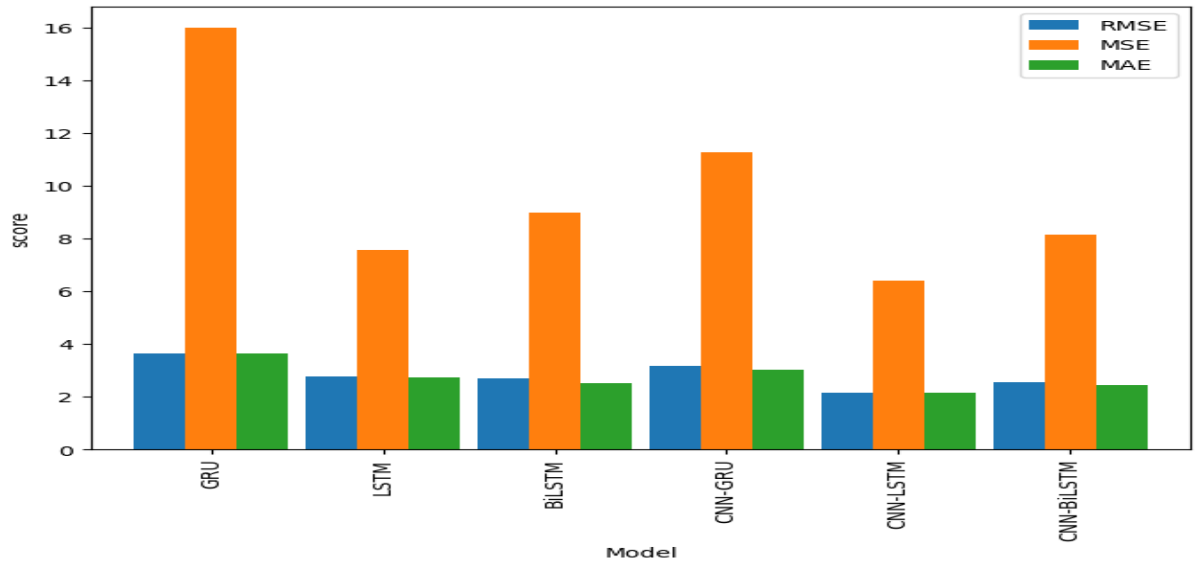


Figure 6-4. Forecasting evaluation before feature-engineering

Figure 6-4 shows the graphical representation of the performances of the six identified regression models before feature engineering was performed on the data they were trained on. The performance of the models' shows slight increase in errors as compared to the performance they had with feature engineering performed.

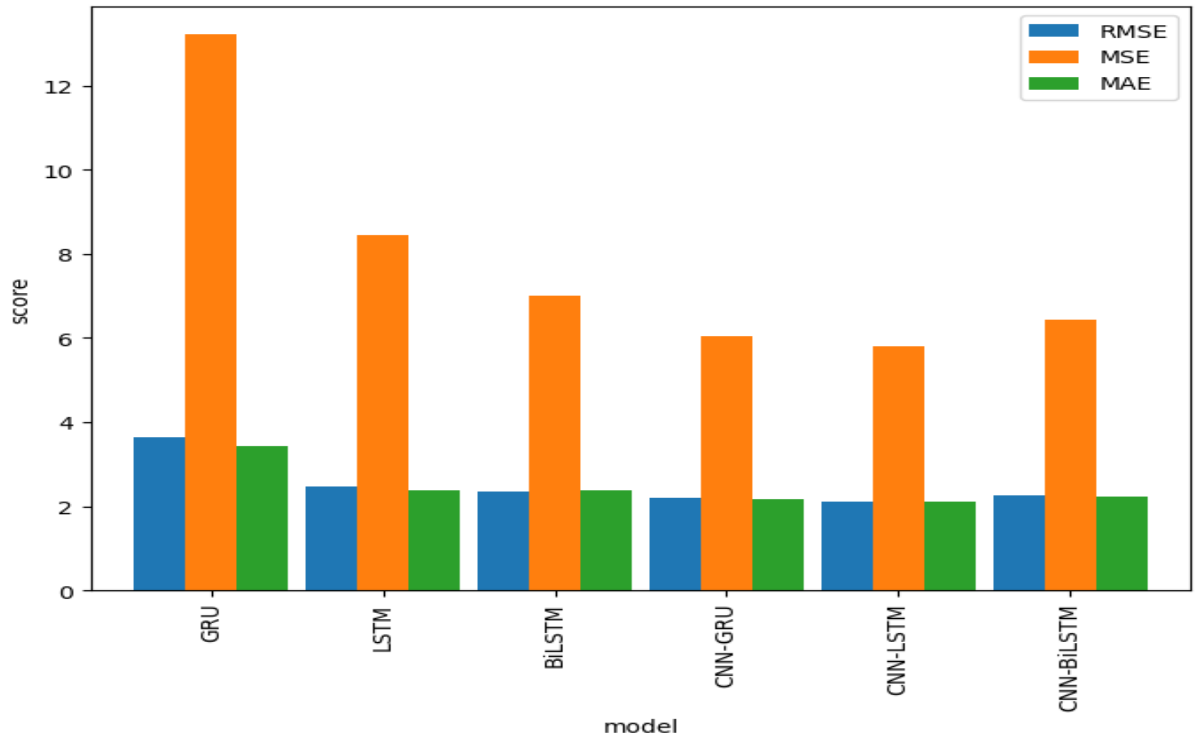


Figure 6-5. Forecasting performance evaluation after feature-engineering

Figure 6-5 shows the graphical representation of the performances of the identified six regression model as evaluated with RMSE, MSE and MAE. As can be observed from the graph the models show slight improvement as compared to before, indicating that the new generated features have allowed the model to develop a slightly better representation of the data and thus enhance its predictive power.

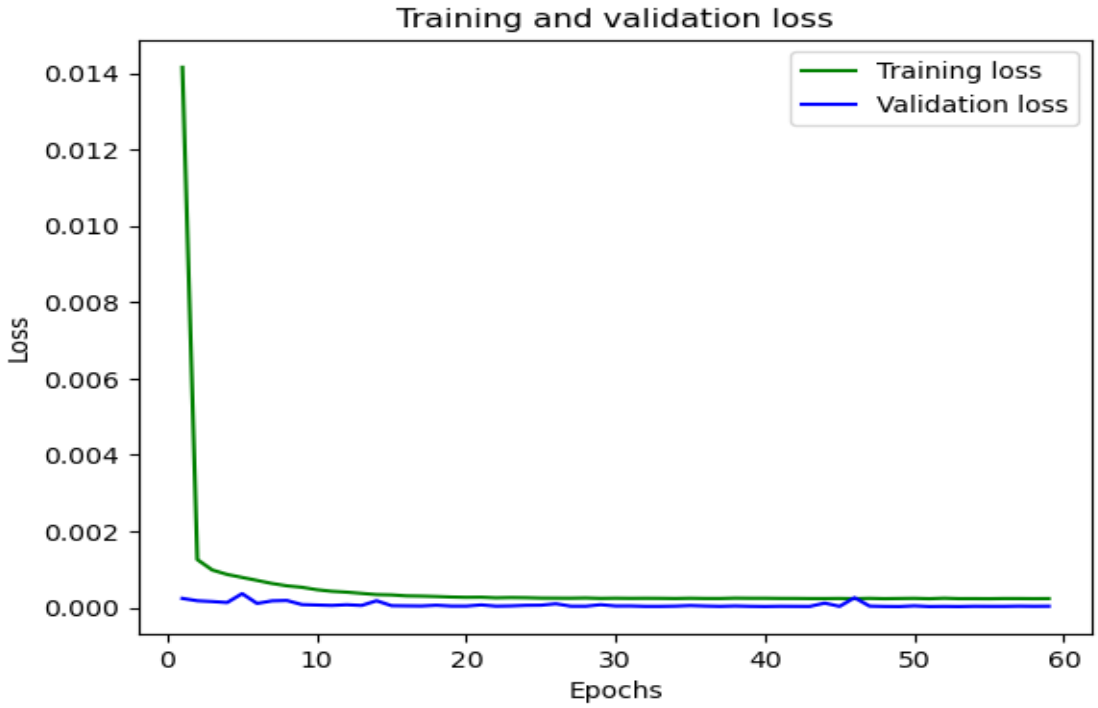


Figure 6-6. Training and validation loss CNN-LSTM

The figure 6-6 show the training and validation loss curve of the CNN-LSTM model. The model was trained with the mse loss function and an early stopping of 10 patience. The graph shows that the model does not overfit or underfit and converges between epochs 50 to 60.

6.3.2. Classification Analysis Results

For classification we built 3 different model architecture for our experiment i.e., MLP, CNN and CNN-MLP and evaluated them against the benchmark metrics i.e., accuracy, precision, recall and f1. We used binary transformation approach for multi-class classification so 4 classifiers were trained for each model since we have 4 classes of faults. Below the results for each classifier for each model is discussed along with their confusion matrix and learning curves.

6.3.2.1. K-Fold cross validation

MLP

K=5	Accuracy(%)	Precision(%)	Recall(%)	F1(%)
Bias	94.70	94.90	94.70	94.77
Drift	99.00	99.10	99.09	99.09
Polydrift	98.80	98.20	98.11	98.11
Random	98.45	98.44	98.43	98.43

K=10	Accuracy(%)	Precision(%)	Recall(%)	F1(%)
Bias	94.78	94.95	94.78	94.77
Drift	99.03	99.17	99.16	99.16
Polydrift	98.81	98.83	98.81	98.81
Random	98.50	98.49	98.48	98.48

Table 6-3. K-Fold cross validation MLP

Table 6-3 show the performance result of the MLP model evaluated using 5 folds and 10 folds cross-validation. Accordingly, it can be observed that the model was able to identify drift, poly-drift and random faults remarkably, but performed less than desired for the bias set.

CNN

K=5	Accuracy(%)	Precision(%)	Recall(%)	F1(%)
Bias	96.00	96.10	96.05	96.05
Drift	99.10	99.20	99.19	99.19
Polydrift	98.60	98.60	98.61	98.61
Random	98.50	98.54	98.50	98.50

K=10	Accuracy(%)	Precision(%)	Recall(%)	F1(%)
Bias	96.05	96.16	96.10	96.10
Drift	99.13	99.30	99.29	99.29

Polydrift	98.63	98.63	98.64	98.64
Random	98.54	98.58	98.54	98.58

Table 6-4. K-Fold cross validation CNN

Table 6-4 show the performance results of CNN model evaluated against 5 folds and 10 folds cross-validation respectively. CNN models are better suited to capturing the spatial features which can be important for identifying relevant patterns in the sensor data, which is why it was able to show slightly better performance on the bias set.

CNN-MLP

K=5	Accuracy(%)	Precision(%)	Recall(%)	F1(%)
Bias	96.10	96.40	96.11	96.09
Drift	99.30	99.30	99.30	99.30
Polydrift	98.80	98.81	98.80	98.80
Random	98.60	98.62	98.60	98.60

K=10	Accuracy(%)	Precision(%)	Recall(%)	F1(%)
Bias	96.11	96.42	96.12	96.10
Drift	99.33	99.33	99.33	99.33
Polydrift	98.81	98.83	98.81	98.81
Random	98.61	98.65	98.61	98.61

Table 6-5. K-Fold cross validation CNN-MLP

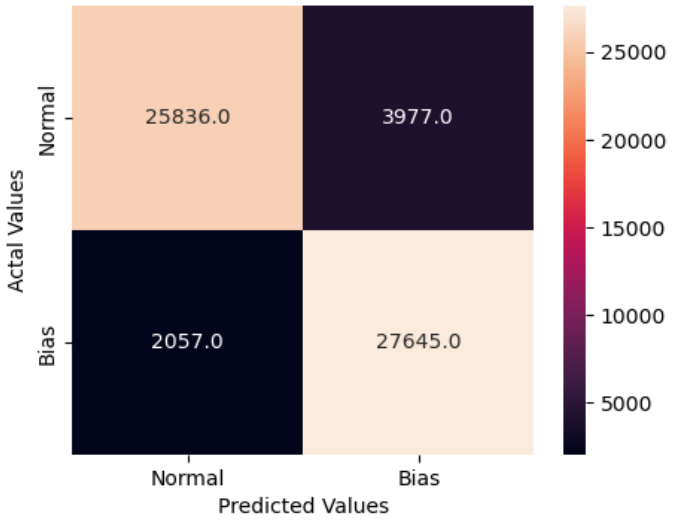
Tables 6-5 shows the performance result of the CNN-MLP model evaluated against 5 folds and 10 folds cross-validation. Accordingly, the CNN-MLP model outperformed both the other models. The combination of CNN and MLP allowed the model to construct a more effective representation of the data, capturing both the spatial and temporal patterns of the data.

6.3.2.2. Confusion matrix visualization

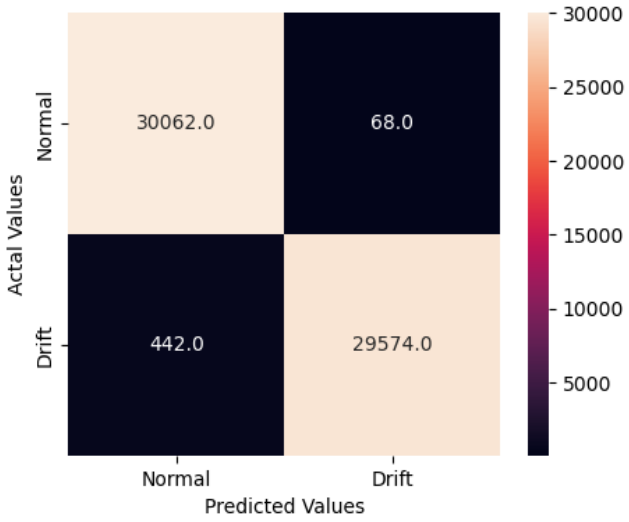
MLP

Figure 6-7 illustrate the confusion matrices for the bias, drift, random, Polydrift and malfunction set experimented with MLP model. For example, for the bias set, from a total of 59,515 samples, the model was able to correctly classify 25,836 as normal and 27,645 as bias. The model incorrectly classified 3,977 normal samples as bias and 2057 bias samples as normal. For the drift set, from a total of 60,146 samples, the model was able to correctly classify 30,062 as normal and 29,574 as drift, and incorrectly classified 68 normal samples as drift and 442 drift samples as normal. For the random set, out of total of 70,137, the model classified correctly all the normal samples and 33,965 random samples correctly. It incorrectly classified 955 random values as normal. For the Polydrift set, out of total of 60,137, 30,072 normal samples were correctly classified with only 55 samples incorrectly classified as Polydrift. 29,314 Polydrift values were correctly classified with only 696 samples incorrectly classified as normal.

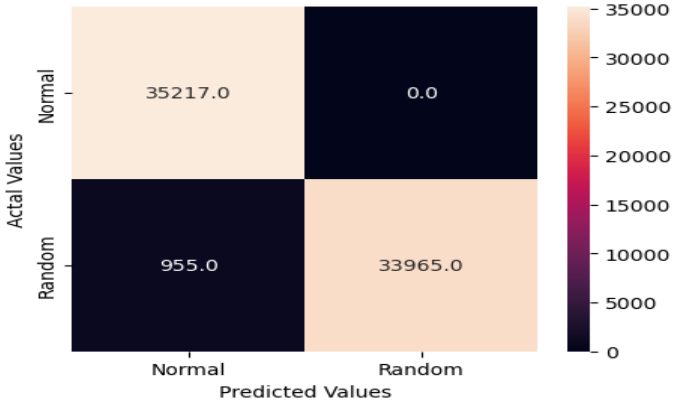
confusion matrix for bias



confusion matrix for drift



confusion matrix for random



confusion matrix for Polydrift

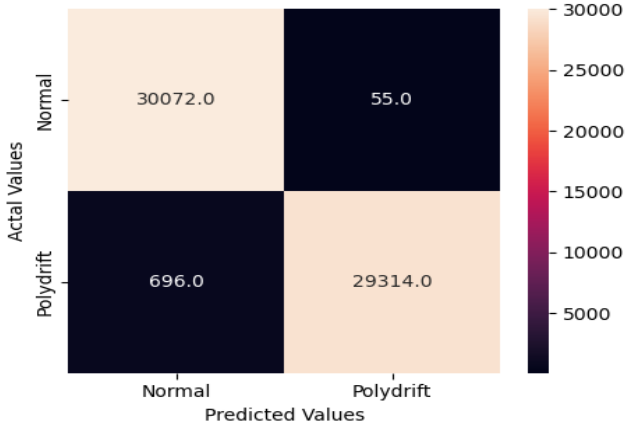
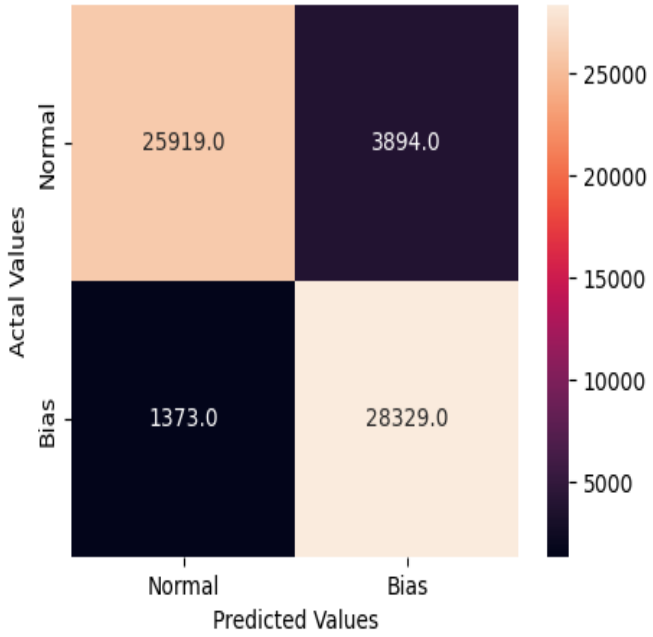


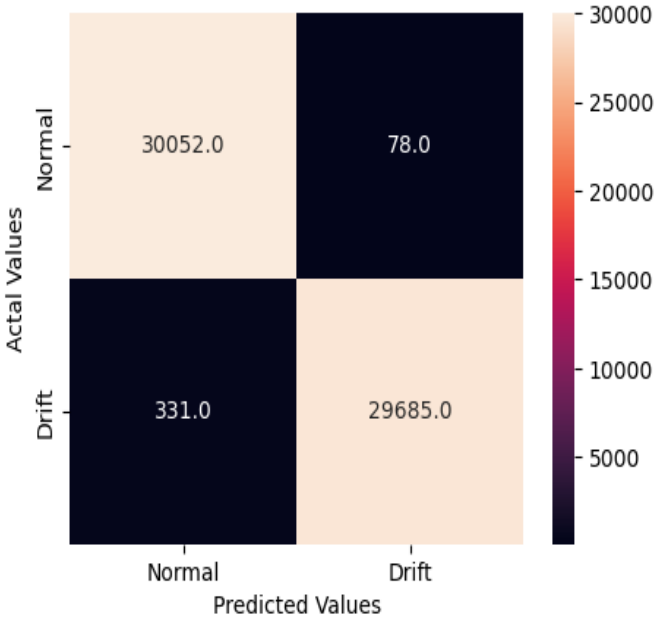
Figure 6-7. MLP Confusion matrix

CNN

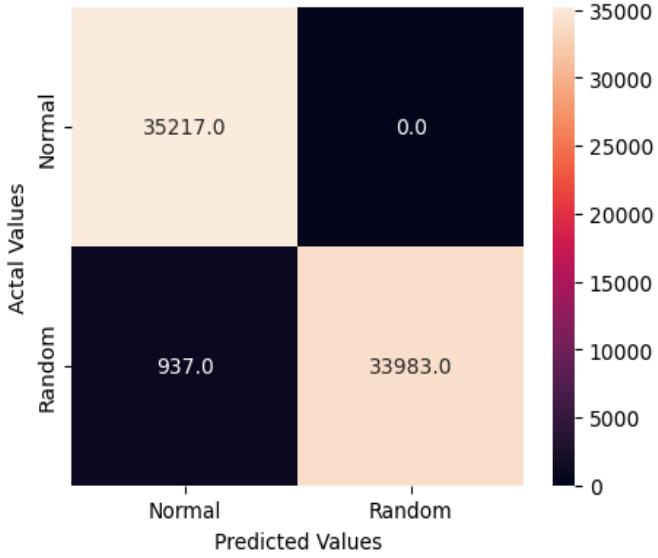
confusion matrix for bias



confusion matrix for drift



confusion matrix for random



confusion matrix for Polydrift

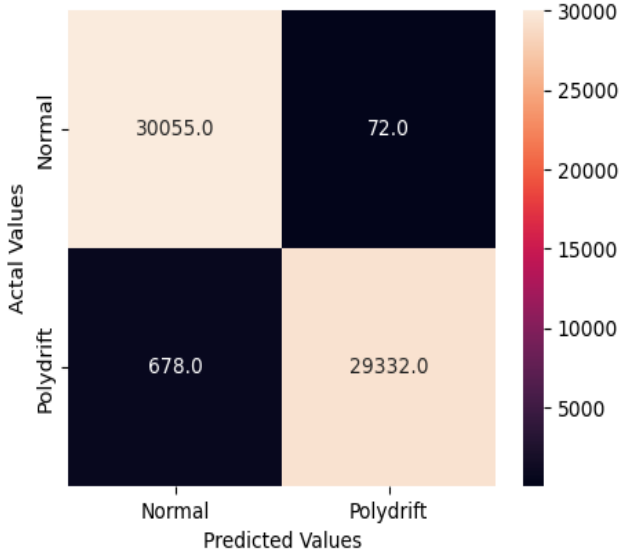
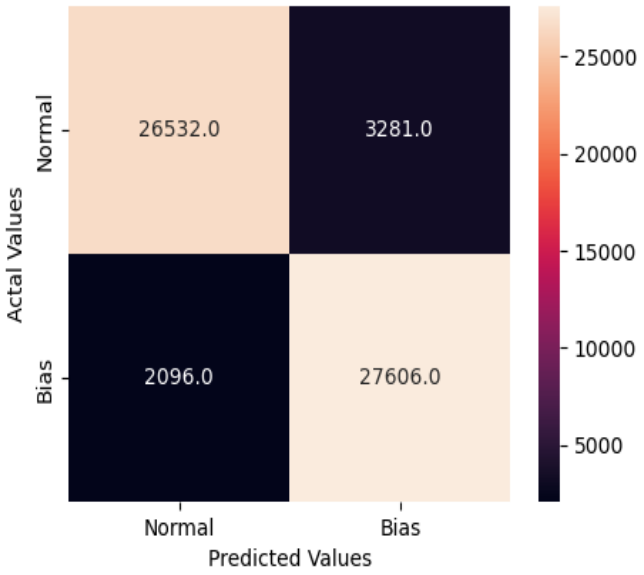


Figure 6-8. CNN Confusion matrix

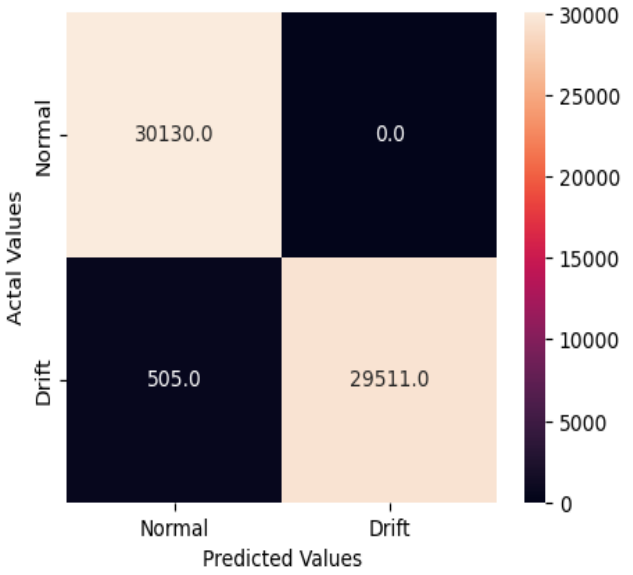
Figure 6-8 illustrate the performance of CNN model using confusion matrices. For example, for the bias set, out of a total of the 59,515 samples, 25,919 were correctly classified as normal and 28,329 were correctly classified as bias. 3,894 normal samples were incorrectly classified as bias and 1,373 bias samples were incorrectly classified as normal. For the drift set, out of the total 60,146 samples, 30052 were correctly classified as normal and 29,685 were correctly classified as drift. 78 normal samples were incorrectly classified as drift and 331 drift samples incorrectly classified as normal.

CNN-MLP

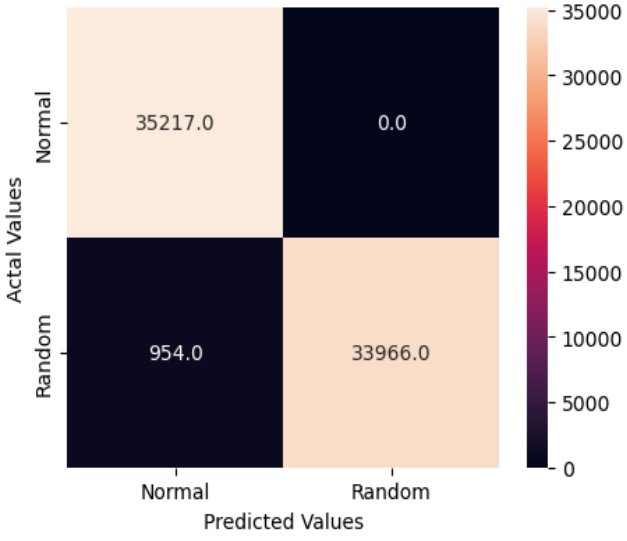
confusion matrix for bias



confusion matrix for drift



confusion matrix for random



confusion matrix for Polydrift

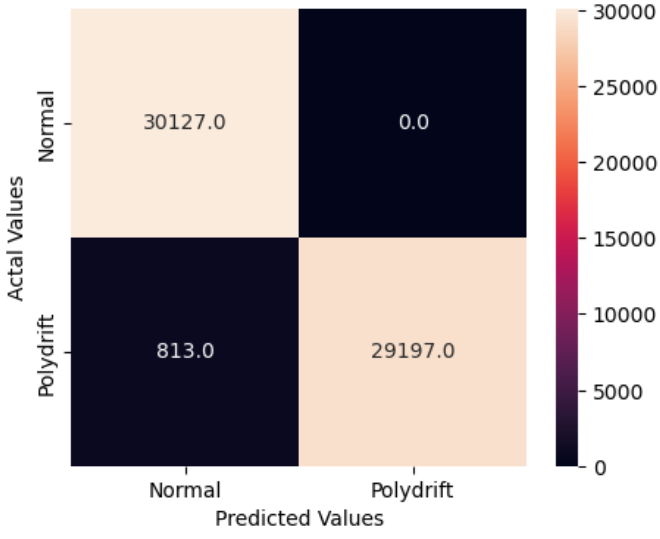


Figure 6-9. CNN-MLP confusion matrix

Figure 6-9 shows the performance of the CNN-MLP model using the confusion matrices. Accordingly, for the bias set, out of the total 59,515 samples, 26532 were correctly classified as normal samples and 27,606 correctly classified as bias samples. 3281 normal samples were incorrectly classified as bias and 2096 bias samples classified as normal. For the drift set, out of the total 60,146 samples all normal samples were correctly classified and 29,511 drift samples were correctly classified. 505 drift samples were incorrectly classified as normal. For the random set, out of the total 70,137 samples, all normal samples were correctly classified and 33966 samples were correctly classified as random. 954 random samples were incorrectly classified as normal.

6.3.2.3. CNN-MLP learning curve

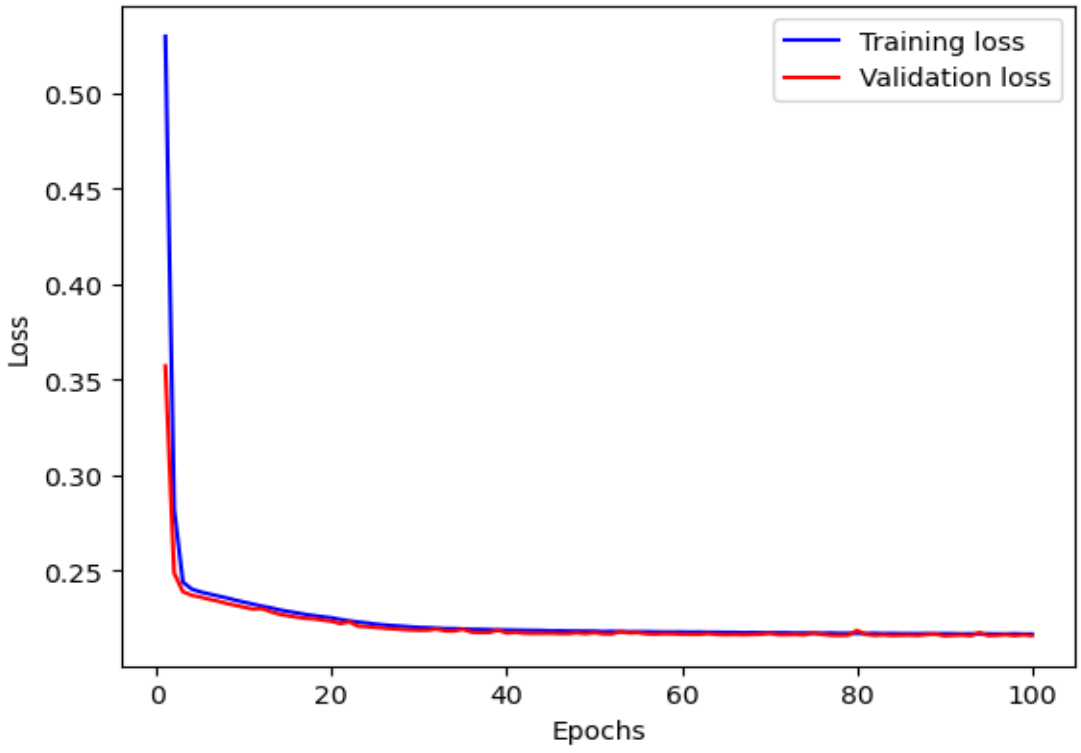


Figure 6-10. CNN-MLP Training and validation loss for Bias fault

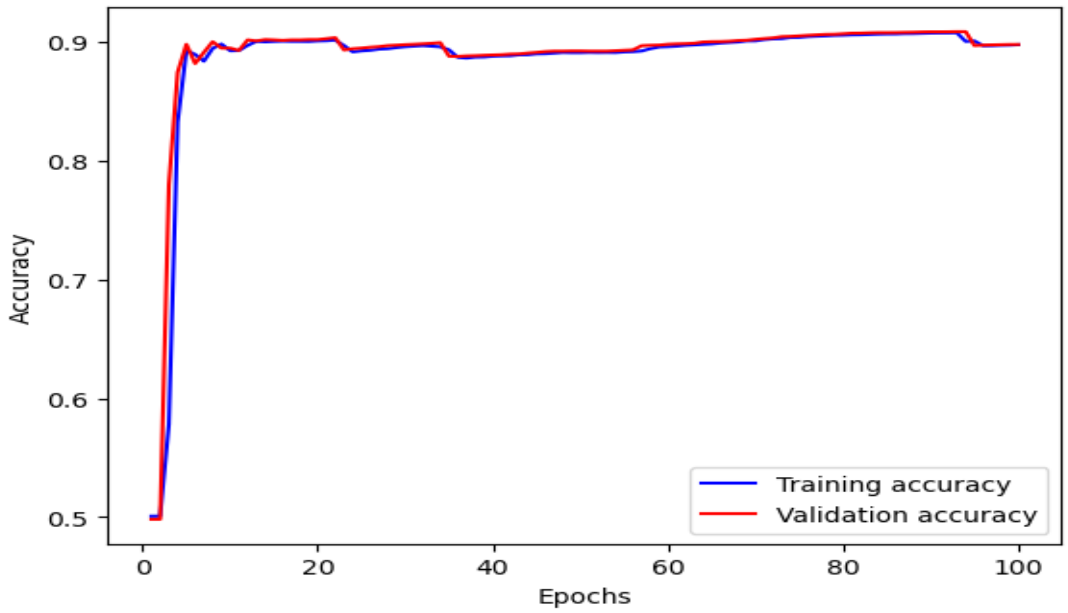


Figure 6-11. CNN-MLP Training and validation accuracy for Bias fault

Figures 6-10 and 6-11 shows the training and validation loss and accuracy curve of the CNN-MLP model trained on the bias set. The model was trained using binary_crossentropy loss function with an early stopping of 10 patience. The graph shows that the training stopped at the 100th epoch as no improvement has been recorded for the previous 10 consecutive epochs.

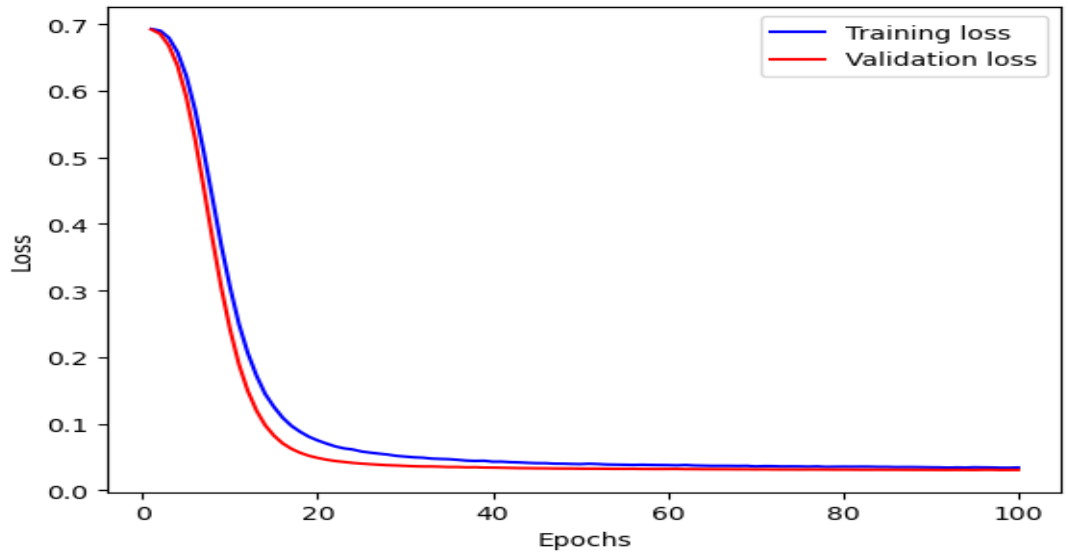


Figure 6-12. CNN-MLP Training and validation loss for Drift fault

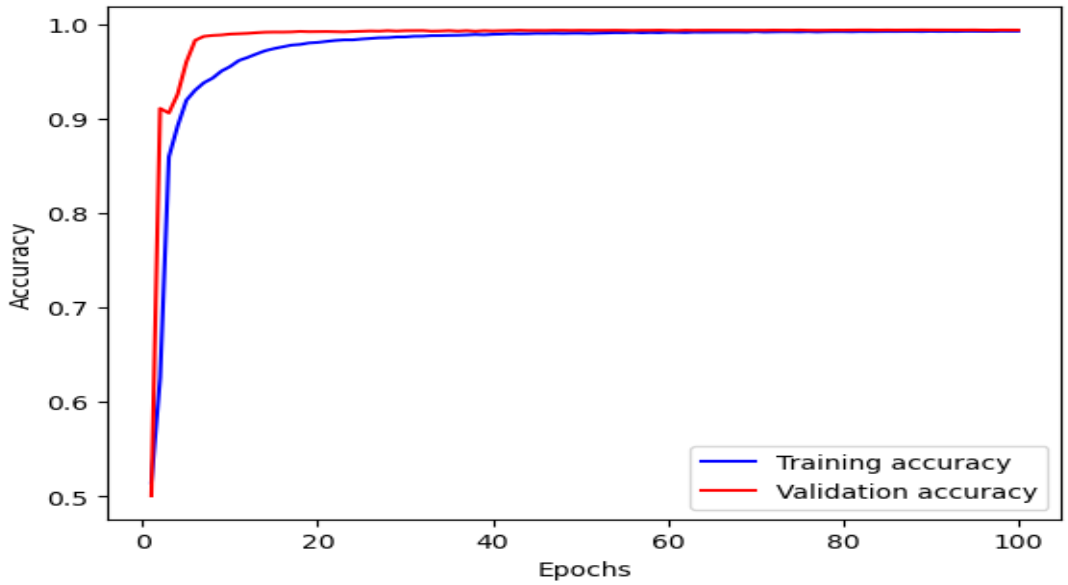


Figure 6-13. CNN-MLP Training and validation accuracy for Drift fault

Figures 6-12 and 6-13 shows the training and validation loss and accuracy curve of the CNN-MLP model trained on the drift set. The model was trained using binary_crossentropy loss function with an early stopping of 10 patience. The graph shows that the training stopped at the 100th epoch as no improvement has been recorded for the previous 10 consecutive epochs.

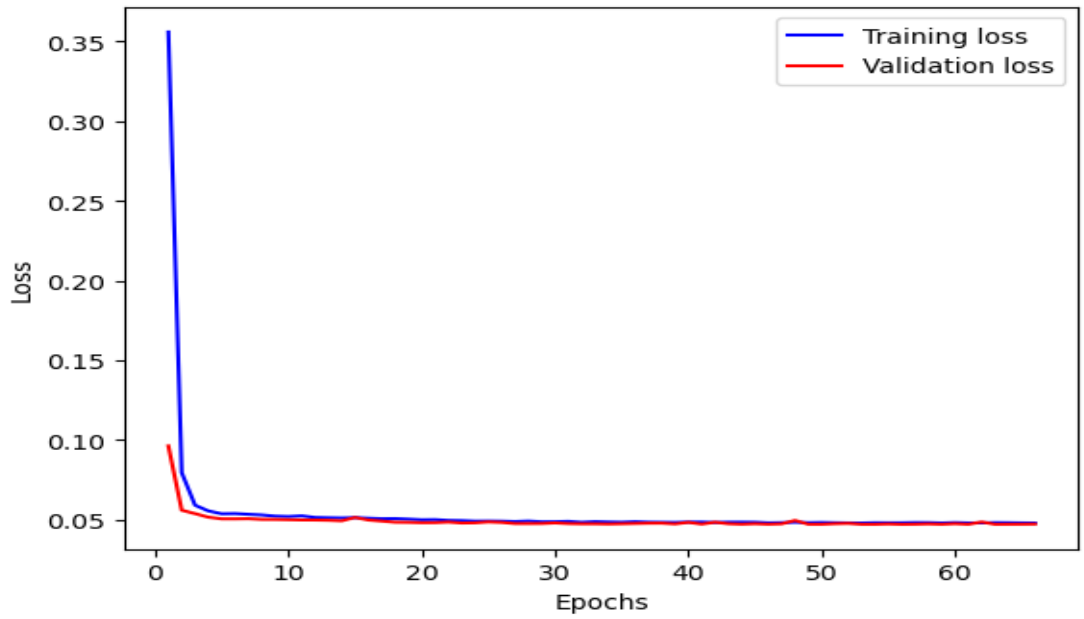


Figure 6-14. CNN-MLP Training and validation loss for Poly-drift fault

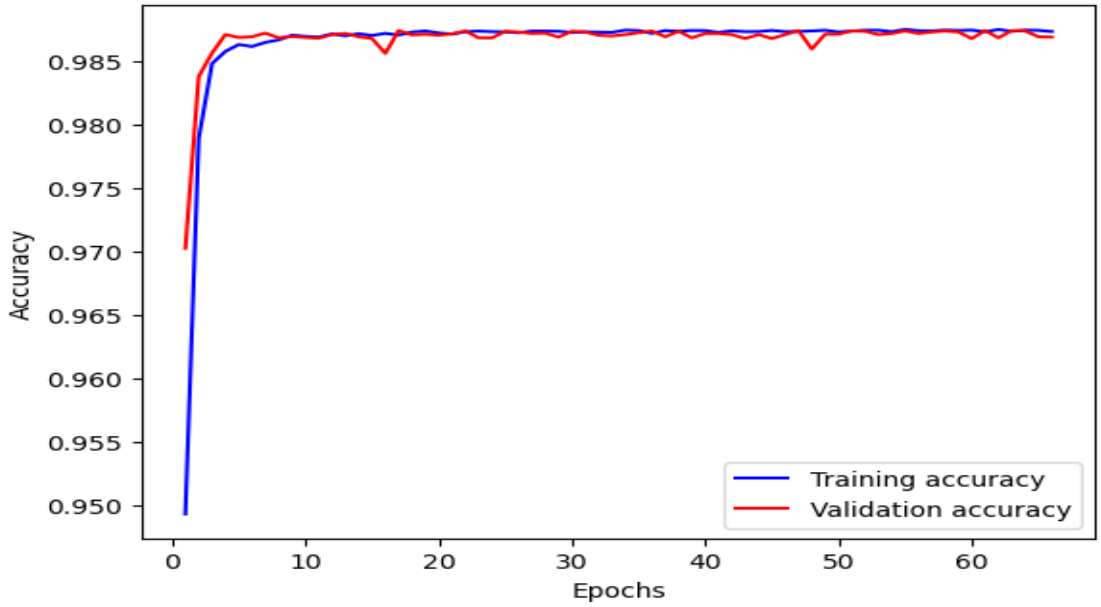


Figure 6-15. CNN-MLP Training and validation accuracy for Poly-drift fault

Figures 6-14 and 6-15 shows the training and validation loss and accuracy curve of the CNN-MLP model trained on the poly-drift set. The model was trained using binary_crossentropy loss function with an early stopping of 10 patience. The graph shows that the training stopped at the 30th epoch as no improvement has been recorded for the previous 10 consecutive epochs.

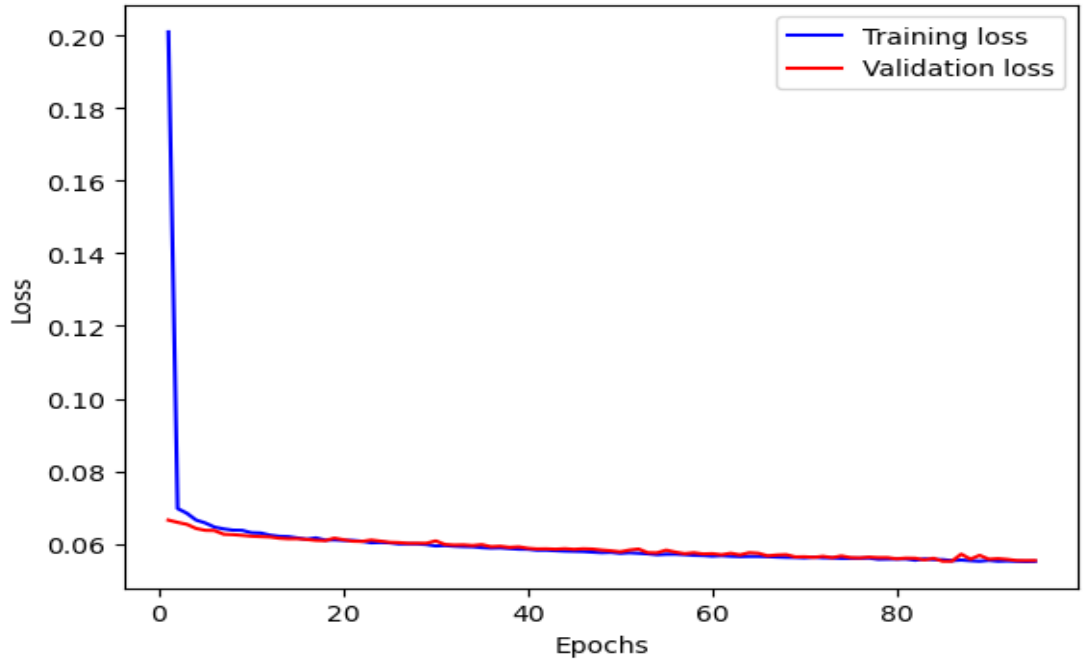


Figure 6-16. CNN-MLP Training and validation loss for Random fault

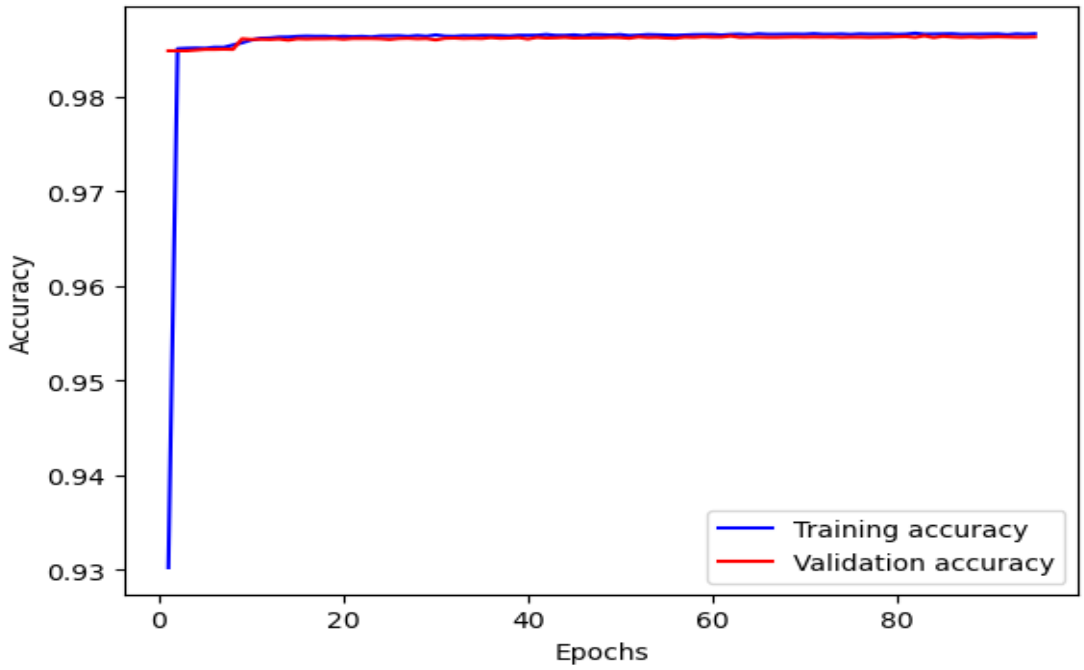


Figure 6-17. CNN-MLP Training and validation accuracy for Random fault

Figures 6-16 and 6-17 shows the training and validation loss and accuracy curve of the CNN-MLP model trained on the random set. The model was trained using binary_crossentropy loss function with an early stopping of 10 patience. The graph shows that the training stopped at the 100th epoch as no improvement has been recorded for the previous 10 consecutive epochs.

6.4. Hyper-parameter Tuning Results

#	Pool size	Units	Learning rate	Kernel size	Filters	Dropout rate	Split_0 score	Split_1 score	Mean score
0	2	128	0.1	3	16	0.5	-31.645	-44.172	-37.909
1	2	128	0.1	2	64	0.4	-0.110	-0.841	-0.476
2	2	64	0.1	3	32	0.4	-8.304	-0.074	-4.189
3	2	32	0.1	3	64	0.5	-25.565	-3.813	-14.689
4	2	128	0.001	3	16	0.4	-40.345	-0.967	-20.656
5	2	64	0.01	2	32	0.4	-0.345	-33.769	-17.057
6	2	32	0.001	2	64	0.4	-0.089	-0.297	-0.193
7	2	128	0.001	3	32	0.5	-0.066	-0.934	-0.500

8	2	64	0.01	3	16	0.4	-13.011	-0.509	-6.760
9	2	128	0.01	3	16	0.4	-0.107	-2.098	-1.102

Table 6-6. Hyper-parameter tuning results

Table 6-6 shows the result of hyper-parameter tuning. We cross validated using TimeSeriesSplit with n_splits of 2 and scoring of neg_mean_squared_error. EarlyStopping with patience of 10 was used to avoid overfitting. The gridsearch method explores all possible combinations. Since this is not a feasible option in terms of time, we used the keras RandomizedSearchCV class with n_iter of 10.

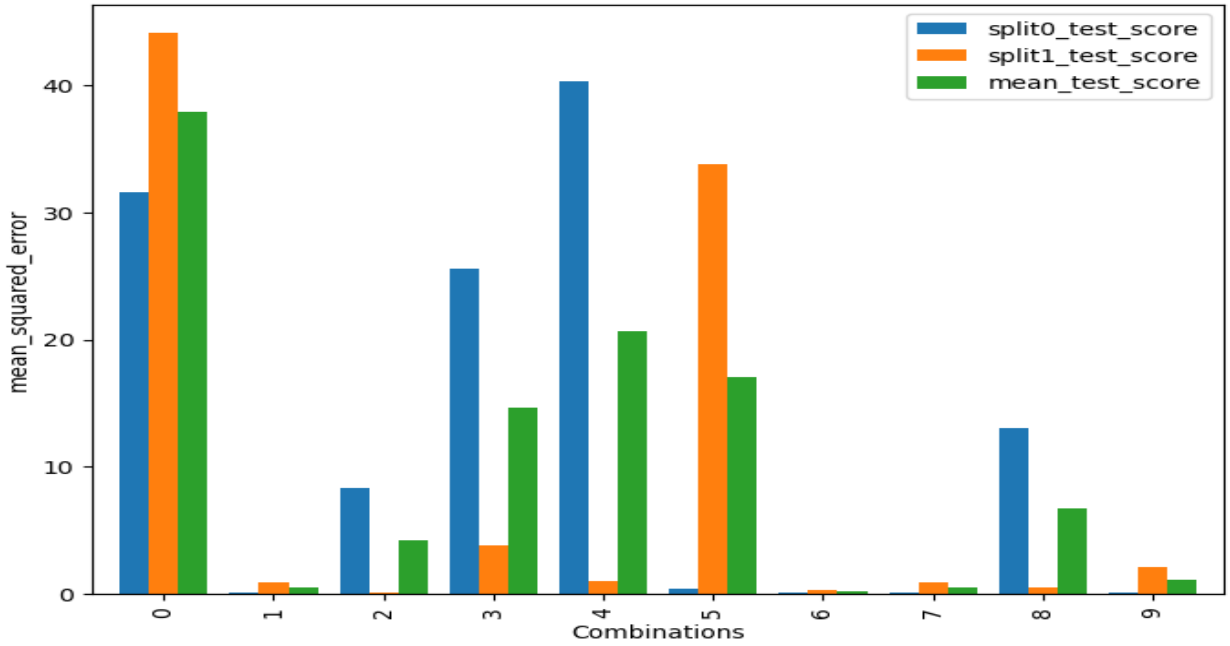


Figure 6-18. Hyper-parameter tuning performance evaluation

Figure 6-18 shows the graphical representation of the 10 combination of hyper-parameter tuning performed. The result shows both the score of the first split, second split and then an average of the 2 splits

6.5. Discussion

Inaccurate and corrupt sensor measurements can have serious impact on various application and systems across different domains. It can lead to mis-guided decision making, to disrupting normal operations of systems leading to delays and downtimes resulting in loss of revenue, to

even compromising the safety of individuals in safety critical systems. While fault detection plays a vital role in identifying the faults in the measurements, it is however, a reactive approach that identifies the faults after they have occurred. This reactive nature means that actions are taken after the fault has already impacted the system and potentially leading to some negative consequences. In this study, we aimed to take a proactive approach that predicts these faults before they occur and thus provide an opportunity for preventive measures.

In this study we performed five experiments to answer the research questions we raised as well as implement our proposed solution.

6.5.1 Experiment 1

In this experiment we performed hyper-parameter tuning. Hyper-parameter tuning refers to the process of selecting the best hyperparameters for the model to be trained with. Different hyper-parameter settings can vary in different model performances and so tuning allow us to find the optimal combination. In this study we used the keras RandomSearchCV class with 10 different combinations. Accordingly, the combination of pool_size = 2, units =32, learning_rate=0.001, kernel_size= 2, filters=64 and Dropout_rate of 0.4 was found to be the best fit with a mean score of 0.193.

6.5.2 Experiment 2

In this experiment we performed feature engineering and selection task. First, we generated both time-based features and rolling window statistics feature generation with a window of 10. The rolling window features generated are rolling mean, rolling variance, rolling std, rolling min and rolling max. Three separate experiments i.e., feature correlation, feature importance ranking using random regressor and feature importance ranking using XGBoost was performed, and in all the cases, the features with highest ranking and correlations were rolling mean, rolling min and rolling max and so these features were selected.

6.5.3 Experiment 3

In this experiment we identified candidate models that are suitable for multi-step forecasting sensor values and evaluated each of them using regression metrics RMSE, MSE and MAE. We divided this experiment into two sub-experiments.

Experiment 3(a).

Under this experiment we evaluate and tested 3 models i.e., GRU, LSTM and BiLSTM using the TimeSeriesSplit cross validation with 2 splits.

Experiment 3(b).

Under this experiment, to leverage the strength of CNN model to extract features, we incorporated the CNN architecture into the above 3 models (*exp3a)* and designed 3 hybrid models CNN-GRU, CNN-LSTM and CNN-BiLSTM and evaluated the performance of each of them again using the TimeSeriesSplit cross validation with 2 splits.

In the first part of the experiment, BiLSTM performed better than the others with a MAE of 2.3324 forecasted over 45 timesteps, and LSTM came in second with MAE of 2.3862. In the second part of the experiment, after the hybridization, all of the 3 models showed improvements, with CNN-LSTM showing the highest improvement of MAE 2.0957 and CNN-BiLSTM following up with 2.2442. Generally, LSTM, BiLSTM and hybridization of these models were able to perform good with a MAE of $\pm$ 0.1 difference as opposed to GRU or CNN-GRU because of their long-term temporal dependency handling ability. These models can be considered as good models for timeseries forecasting thus answering RQ1. The second part of the experiment also addresses RQ2 by building 3 instances of hybrid models by leveraging the power of CNN and thus demonstrating how hybrid models can be built.

6.5.4. Experiment 4

In this experiment we identified three candidate models for sensor fault classification task and evaluated the models using evaluation metrics accuracy, precision, recall and f1 with 5- and 10-folds cross validation. The identified models were CNN, MLP and CNN-MLP. Since for the multi-class problem we used binary transformation, for each fault a classifier was trained and the classifier with the highest probability was selected. This is done for all the three models. Accordingly, for both the 5-and 10-folds cross validation, CNN-MLP outperformed the other two with an accuracy of 91.11% for bias, 99.33% for drift, 98.81% for poly-drift and 98.61% for random using the 10 folds cross validation. The reason for this can be attributed to the fact that it leveraged both of the strengths of the other models. This experiment also addresses RQ2 as the implementation of CNN-MLP illustrates how we can design a hybrid model that leverages the strength of different models for a particular task.

6.5.5. Experiment 5

Under this experiment we trained and evaluated the models identified for forecasting with only the raw data without adding any generated rolling window statistics features and then compared it against models trained and evaluated with generated features incorporated into them and thus evaluated the impact of the new incorporated features. Accordingly, models with additional features showed a minor increase in their performance. This is because by performing feature engineering where we generated new rolling window features such as rolling mean, rolling maximum, and rolling minimum, we were able to provide additional information about the local patterns and variations within the data, making the underlying patterns and trends more apparent. The rolling window features also provided a compact representation of the time series data thus reducing the dimensionality of the input space, making it more manageable for the model to process and extract relevant information. Accordingly, GRU showed a decrease of 0.21 in MAE, LSTM -0.33, Bi-LSTM -0.19, CNN-GRU -0.85, CNN-LSTM -0.04 and CNN-BiLSTM -0.17. This goes to show that although the difference in their performance in this case is minor (may be the dataset being not as complex), such feature engineering could contribute to the models' predictive powers. And with this experiment we were able to answer RQ3.

Models	RMSE	MSE	MAE
GRU	**3.6353**	**13.2154**	**3.4260**
LSTM	**2.4569**	**8.4576**	**2.3862**
BiLSTM	**2.3558**	**7.0005**	**2.3324**
CNN-GRU	**2.2061**	**6.0298**	**2.1585**
CNN-LSTM	**2.1037**	**5.7972**	**2.0957**
CNN-BiLSTM	**2.2522**	**6.4196**	**2.2442**

Table 6-7. Summary of regression performance evaluation

Table 6-7 shows the summary of the performances of the regression model under the evaluation metrics of rmse, mse and mae. Accordingly, from the identified 6 models, it can be seen that CNN-LSTM was able to outperform the others.

		Accuracy	Precision	Recall	F1
Bias	MLP	94.78	94.95	94.78	94.77
	CNN	96.05	96.16	96.10	96.10
	CNN-MLP	96.11	96.42	96.12	96.10
Drift	MLP	99.03	99.17	99.16	99.16
	CNN	99.13	99.30	99.29	99.29
	CNN-MLP	99.33	99.33	99.33	99.33
Random	MLP	98.50	98.49	98.48	98.48
	CNN	98.54	98.58	98.54	98.58
	CNN-MLP	98.61	98.65	98.61	98.61
Poly-drift	MLP	98.81	98.83	98.81	98.81
	CNN	98.63	98.63	98.64	98.64
	CNN-MLP	98.81	98.83	98.81	98.81

Table 6-8. Summary of classification performance evaluation

Table 6-8 shows a summary of the performance of the classification models. Accordingly, from the evaluated 3 models, it can be seen that CNN-MLP outperformed the others. This can be attributed to the fact that it was able to leverage the strength of both models.

6.5.6. Baseline Evaluation

The baseline paper (Emperuman & Chandrasekaran, 2020) evaluated their model (CDHMM-LVQ) against 10%, 20%, 30%, 40% and 50% injection rate and performed the classification for the faults bias, drift, random and spike. Since the injection rate for our dataset was 21% for bias and for drift, random and poly-drift 20%, we used their performance of the 20% rate to compare our model against. Accordingly, for the 20% rate they achieved, bias 98.33%, drift 96.67%, spike 96.67% and random 100%, which gives an average of 97.91% accuracy. Our model achieved bias 96.11%, drift 99.33%, poly-drift 98.81% and random 98.61%, scoring an average accuracy of 98.21% and thus showing an improvement of 0.3% of the baseline work.

CHAPTER SEVEN

CONCLUSIONS and FUTURE WORKS

7.1. Overview

In this chapter we will revise some of the key points discussed throughout the study, highlight key findings and limitations of the study under the conclusion section. We also put forward future works that could be addressed under recommendation section.

7.2. Conclusions

In this study, we reviewed sensor faults, their causes and types, their impacts on the different industries and the approaches used for detecting them throughout the literature. We found that though many works have been done on identifying and diagnosing these faults, most of them focus on detecting the faults. While detecting a fault is useful it is a reactive approach that addresses the problem after it has occurred. It is more beneficial to predict the potential occurrence of a fault before it happens, especially if that fault can lead the system to total failure. This can prevent costly downtime, reduce maintenance costs, and increase the safety of critical systems.

In this study we developed a proactive approach that predicts the potential occurrence of faults using hybrid deep learning models. The solution involves a two-stage procedure where in the first stage, a hybrid CNN-LSTM model is trained on a clean non-faulty historical data of the sensors to perform a multi-step forecasting of future values. In the second stage, a hybrid CNN-MLP model is trained on fault injected sensor data to learn the patterns of the different fault types and classify new data accordingly. The forecasted values of the first hybrid model are then passed as input to the second hybrid model to determine its status.

For the regression task, we experimented with different forecasting models like LSTM, GRU, BiLSTM, CNN-LSTM, CNN-BiLSTM and CNN-GRU both before and after incorporating rolling window statistical features. We evaluated each of them using evaluation metrics RMSE, MSE and MAE and finally chose CNN-LSTM with incorporated rolling window features as it outperformed the other models with a MAE of 2.0957 for a 45 timesteps forecast. For the classification task, we experimented with MLP, CNN and CNN-MLP and evaluated them using the classification evaluation metrics accuracy, precision, recall and f1 using 5-folds

and 10-folds cross validation and in both cases CNN-MLP outperformed the other models with accuracy of 96.11% for bias, 99.33% for drift, 98.61% for random and 98.81% for poly-drift, an average of 98.21% over the 4 faults. Finally, we compared our work with the baseline work and achieved a 0.3% increase in the average accuracy across the 4 faults.

7.3. Contributions of the study

- The development of a multi-stage approach for sensor fault prediction and classification using hybrid deep learning models.
- A comparative analysis of GRU, LSTM, BiLSTM, CNN-GRU, CNN-LSTM, CNN-BiLSTM for multi-step univariate timeseries forecasting.
- An analysis of the impact of feature engineering on the predictive performance of timeseries models.
- A literature survey of the different fault types, their impacts and the different approaches applied to identify them.

7.4. Application of the study

The proposed solution can be deployed and integrated at the different level of the IoT ecosystem. For example:

- Cloud platforms: IoT cloud platforms are responsible for storing, processing, and analyzing vast amounts of sensor data. The models can be integrated into these cloud platforms, where it can be used to preprocess the incoming sensor data, predict and classify faults, and generate alerts or notifications for system administrators or stakeholders.
- Gateway devices: In IoT architectures, gateway devices act as intermediaries between edge devices and cloud platforms. These gateways can host the models and perform fault prediction and classification tasks on the collected sensor data, allowing for analysis of the data before transmitting it to the cloud for further analysis.
- Edge devices: IoT edge devices are often equipped with sensors that collect data from the surrounding environment. Although these devices typically have limited processing, memory and storage capabilities, the models can be deployed directly on these edge devices to perform prediction and classification of sensor data faults.

In all these and other cases, by integrating the model at different levels (edge, gateway, cloud), the IoT system can have the ability to take proactive measures, minimize downtime, optimize decision-making, and enhance overall system reliability and efficiency.

7.5. Future works

In this study due to lack of data we were limited to only historical data of the sensors. For the future researchers we recommend the following:

- Incorporate additional features and enhance the predictive power of the forecasting model so that longer timesteps can be forecasted with lesser errors.
- Generalize the system to also work for system centric sensor faults.
- Generalize the system so that it works for other types of sensors as well.

SPECIAL ACKNOWLEDGMENTS

This research was funded by Adama Science and Technology University with grant number: ASTU/ SM-R/831/23

Adama, Ethiopia

References

Abouelela, A., Abbas, H., Eldeeb, H., & Nassar, S. (2000). Statistical approach for textile fault detection. *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, *4*, 2857–2862. https://doi.org/10.1109/ICSMC.2000.884431

Ali, F., Ali, A., Imran, M., Naqvi, R. A., Siddiqi, M. H., & Kwak, K. S. (2021). Traffic accident detection and condition analysis based on social networking data. *Accident Analysis & Prevention*, *151*, 105973. https://doi.org/10.1016/J.AAP.2021.105973

Alwan, A. A., Brimicombe, A. J., Ciupala, M. A., Ghorashi, S. A., Baravalle, A., & Falcarin, P. (2022). Time-series clustering for sensor fault detection in large-scale Cyber–Physical Systems. *Computer Networks*, *218*, 109384. https://doi.org/10.1016/J.COMNET.2022.109384

Biddle, L., & Fallah, S. (2021). A Novel Fault Detection, Identification and Prediction Approach for Autonomous Vehicle Controllers Using SVM. *Automotive Innovation*, *4*(3), 301–314. https://doi.org/10.1007/S42154-021-00138-0/FIGURES/8

Bouktif, S., Fiaz, A., Ouni, A., & Serhani, M. A. (2018). Optimal Deep Learning LSTM Model for Electric Load Forecasting using Feature Selection and Genetic Algorithm: Comparison with Machine Learning Approaches †. *Energies 2018, Vol. 11, Page 1636*, *11*(7), 1636. https://doi.org/10.3390/EN11071636

Clouqueur, T., Ercevik, O., Saluja, K. K., & Takahashi, H. (2001). Efficient signature-based fault diagnosis using variable size windows. *Proceedings of the IEEE International Conference on VLSI Design*, 391–396. https://doi.org/10.1109/ICVD.2001.902690

De Bruijn, B., Nguyen, T. A., Bucur, D., & Tei, K. (2016). Benchmark datasets for fault detection and classification in sensor data. *SENSORNETS 2016 - Proceedings of the 5th International Confererence on Sensor Networks*, 185–195. https://doi.org/10.5220/0005637901850195

Djeziri, M. A., Benmoussa, S., & Sanchez, R. (2018). Hybrid method for remaining useful life prediction in wind turbine systems. *Renewable Energy*, *116*, 173–187. https://doi.org/10.1016/J.RENENE.2017.05.020

Emperuman, M., & Chandrasekaran, S. (2020). Hybrid continuous density hmm-based ensemble neural networks for sensor fault detection and classification in wireless sensor network. *Sensors (Switzerland)*, *20*(3). https://doi.org/10.3390/S20030745

Eseye, A. T., Lehtonen, M., Tukia, T., Uimonen, S., & John Millar, R. (2019). Machine learning based integrated feature selection approach for improved electricity demand forecasting in decentralized energy systems. *IEEE Access*, *7*, 91463–91475. https://doi.org/10.1109/ACCESS.2019.2924685

Gao, Z., Cecati, C., & Ding, S. X. (2015). A survey of fault diagnosis and fault-tolerant techniques-part I: Fault diagnosis with model-based and signal-based approaches. *IEEE Transactions on Industrial Electronics*, *62*(6), 3757–3767. https://doi.org/10.1109/TIE.2015.2417501

Habibi, H., Howard, I., & Simani, S. (2019). Reliability improvement of wind turbine power generation using model-based fault detection and fault tolerant control: A review. *Renewable Energy*, *135*, 877–896. https://doi.org/10.1016/J.RENENE.2018.12.066

Hashimoto, M., Kawashima, H., & Oba, F. (2003). A Multi-Model Based Fault Detection and Diagnosis of Internal Sensor for Mobile Robot. *IEEE International Conference on Intelligent Robots and Systems*, *4*, 3787–3792. https://doi.org/10.1109/IROS.2003.1249744

He, C., Zhang, X., & Jia, B. (2013). UIO based robust fault diagnosis approach for aero-engine fiber-optic sensor. *IEEE International Conference on Automation Science and Engineering*, 550–553. https://doi.org/10.1109/COASE.2013.6653947

Hiransha, M., Gopalakrishnan, E. A., Menon, V. K., & Soman, K. P. (2018). NSE Stock Market Prediction Using Deep-Learning Models. *Procedia Computer Science*, *132*, 1351–1362. https://doi.org/10.1016/J.PROCS.2018.05.050/NSE_STOCK_MARKET_PREDICTION_USING_DEEP_LEARNING_MODELS.PDF

Huang, J., Li, M., Zhang, Y., Mu, L., Ao, Z., & Gong, H. (2021). Fault Detection and Classification for Sensor Faults of UAV by Deep Learning and Time-Frequency Analysis. *Chinese Control Conference, CCC*, *2021-July*, 4420–4424. https://doi.org/10.23919/CCC52363.2021.9550141

Jana, D., Patil, J., Herkal, S., Nagarajaiah, S., & Duenas-Osorio, L. (2022). CNN and Convolutional Autoencoder (CAE) based real-time sensor fault detection, localization, and correction. *Mechanical Systems and Signal Processing*, *169*, 108723. https://doi.org/10.1016/J.YMSSP.2021.108723

Javaid, M., Haleem, A., Rab, S., Pratap Singh, R., & Suman, R. (2021). Sensors for daily life: A review. *Sensors International*, *2*, 100121. https://doi.org/10.1016/J.SINTL.2021.100121

Jihani, N., Kabbaj, M. N., & Benbrahim, M. (2023). Sensor fault detection and isolation for smart irrigation wireless sensor network based on parity space. *International Journal of Electrical and Computer Engineering (IJECE)*, *13*(2), 1463–1471. https://doi.org/10.11591/IJECE.V13I2.PP1463-1471

Kim, S., & Kim, H. (2016). A new metric of absolute percentage error for intermittent demand forecasts. *International Journal of Forecasting*, *32*(3), 669–679. https://doi.org/10.1016/J.IJFORECAST.2015.12.003

Li, D., Wang, Y., Wang, J., Wang, C., & Duan, Y. (2020). Recent advances in sensor fault diagnosis: A review. *Sensors and Actuators A: Physical*, *309*, 111990. https://doi.org/10.1016/J.SNA.2020.111990

Liu, W. X., Yin, R. P., & Zhu, P. Y. (2022). Deep Learning Approach for Sensor Data Prediction and Sensor Fault Diagnosis in Wind Turbine Blade. *IEEE Access*, *10*, 117225–117234. https://doi.org/10.1109/ACCESS.2022.3219480

Manzoni, E., Rampazzo, M., & Favero, S. Del. (2021). Detection of Glucose Sensor Faults in an Artificial Pancreas via Whiteness Test on Kalman Filter Residuals. *IFAC-PapersOnLine*, *54*(7), 274–279. https://doi.org/10.1016/J.IFACOL.2021.08.371

Marcellino, M., Stock, J. H., & Watson, M. W. (2006). A comparison of direct and iterated multistep AR methods for forecasting macroeconomic time series. *Journal of Econometrics*, *135*(1–2), 499–526. https://doi.org/10.1016/J.JECONOM.2005.07.020

Mishra, A. K., & Mohanty, S. K. (2022). Intermittent Fault Diagnosis in Sensors using Artificial Neural Network: A Comparative Study. *2022 IEEE International Conference on Current Development in Engineering and Technology (CCET)*, 1–7. https://doi.org/10.1109/CCET56606.2022.10080532

Muthuramalingam, S., Bharathi, A., Rakesh kumar, S., Gayathri, N., Sathiyaraj, R., & Balamurugan, B. (2019). Iot based intelligent transportation system (iot-its) for global perspective: a case study. *Intelligent Systems Reference Library*, *154*, 279–300. https://doi.org/10.1007/978-3-030-04203-5_13/COVER

Najafi Amin, A., McKee, K., Mazhar, I., Bredin, A., Mullins, B., & Howard, I. (2019). Acoustic signature based early fault detection in rolling element bearings. *Lecture Notes in Mechanical Engineering*, 415–422. https://doi.org/10.1007/978-3-319-95711-1_41/COVER

Ni, K., Ramanathan, N., Chehade, M. N. H., Balzano, L., Nair, S., Zahedi, S., Kohler, E., Pottie, G., Hansen, M., & Srivastava, M. (2009). Sensor network data fault types. *ACM Transactions on Sensor Networks (TOSN)*, *5*(3), 1–29. https://doi.org/10.1145/1525856.1525863

Rahimilarki, R., Gao, Z., Jin, N., Binns, R., & Zhang, A. (2020). Data-driven Sensor Fault Estimation for the Wind Turbine Systems. *IEEE International Symposium on Industrial Electronics*, *2020-June*, 1211–1216. https://doi.org/10.1109/ISIE45063.2020.9152490

Safavi, S., Safavi, M. A., Hamid, H., & Fallah, S. (2021). Multi-Sensor Fault Detection, Identification, Isolation and Health Forecasting for Autonomous Vehicles. *Sensors (Basel, Switzerland)*, *21*(7). https://doi.org/10.3390/S21072547

Schneider, P., & Xhafa, F. (2022). Anomaly detection: Concepts and methods. *Anomaly Detection and Complex Event Processing over IoT Data Streams*, 49–66. https://doi.org/10.1016/B978-0-12-823818-9.00013-4

Sethi, P., & Sarangi, S. R. (2017). Internet of Things: Architectures, Protocols, and Applications. *Journal of Electrical and Computer Engineering*, *2017*. https://doi.org/10.1155/2017/9324035

Srimani, S., Ghosh, K., & Rahaman, H. (2016). Parametric Fault Detection in Analog Circuits: A Statistical Approach. *Proceedings of the Asian Test Symposium*, 275–280. https://doi.org/10.1109/ATS.2016.55

Subbaraj, P., & Kannapiran, B. (2010). Artificial Neural Network Approach for Fault Detection in Pneumatic Valve in Cooler Water Spray System. *International Journal of Computer Applications*, *9*(7), 43–52. https://doi.org/10.5120/1395-1881

Taieb, S. Ben, & Hyndman, R. J. (2022). *Recursive and direct multi-step forecasting: the best of both worlds*. https://doi.org/10.26180/21500169.V1

Titouna, C., Naït-Abdesselam, F., & Khokhar, A. (2019). DODS: A Distributed Outlier Detection Scheme for Wireless Sensor Networks. *Computer Networks*, *161*, 93–101. https://doi.org/10.1016/J.COMNET.2019.06.014

Uppal, M., Gupta, D., Juneja, S., Dhiman, G., & Kautish, S. (2021). Cloud-Based Fault Prediction Using IoT in Office Automation for Improvisation of Health of Employees. *Journal of Healthcare Engineering*, *2021*, 1–13. https://doi.org/10.1155/2021/8106467

Uppal, M., Gupta, D., Juneja, S., Sulaiman, A., Rajab, K., Rajab, A., Elmagzoub, M. A., & Shaikh, A. (2022). Cloud-Based Fault Prediction for Real-Time Monitoring of Sensor Data in Hospital Environment Using Machine Learning. *Sustainability 2022, Vol. 14, Page 11667*, *14*(18), 11667. https://doi.org/10.3390/SU141811667

Wahid, A., Breslin, J. G., & Intizar, M. A. (2022). Prediction of Machine Failure in Industry 4.0: A Hybrid CNN-LSTM Framework. *Applied Sciences 2022, Vol. 12, Page 4221*, *12*(9), 4221. https://doi.org/10.3390/APP12094221

Wang, Z. M., Song, G. H., & Gao, C. (2019). An isolation-based distributed outlier detection framework using nearest neighbor ensembles for wireless sensor networks. *IEEE Access*, *7*, 96319–96333. https://doi.org/10.1109/ACCESS.2019.2929581

Welcer, M., Szczepański, C., & Krawczyk, M. (2022). The Impact of Sensor Errors on Flight Stability. *Aerospace 2022, Vol. 9, Page 169*, *9*(3), 169. https://doi.org/10.3390/AEROSPACE9030169

Wibawa, A. P., Utama, A. B. P., Elmunsyah, H., Pujianto, U., Dwiyanto, F. A., & Hernandez, L. (2022). Time-series analysis with smoothed Convolutional Neural Network. *Journal of Big Data*, *9*(1), 1–18. https://doi.org/10.1186/S40537-022-00599-Y/TABLES/12

Xia, F. (2009). Wireless Sensor Technologies and Applications. *Sensors 2009, Vol. 9, Pages 8824-8830*, *9*(11), 8824–8830. https://doi.org/10.3390/S91108824

Xu, J., Gu, B., & Tian, G. (2022). Review of agricultural IoT technology. *Artificial Intelligence in Agriculture*, *6*, 10–22. https://doi.org/10.1016/J.AIIA.2022.01.001

Yamashita, R., Nishio, M., Do, R. K. G., & Togashi, K. (2018). Convolutional neural networks: an overview and application in radiology. *Insights into Imaging*, *9*(4), 611–629. https://doi.org/10.1007/S13244-018-0639-9

Yan, X., Guan, T., Fan, K., & Sun, Q. (2021). Novel double layer BiLSTM minor soft fault detection for sensors in air-conditioning system with KPCA reducing dimensions. *Journal of Building Engineering*, *44*, 102950. https://doi.org/10.1016/J.JOBE.2021.102950

Yang, C., Shen, W., & Wang, X. (2018). The Internet of Things in Manufacturing: Key Issues and Potential Applications. *IEEE Systems, Man, and Cybernetics Magazine*, *4*(1), 6–15. https://doi.org/10.1109/MSMC.2017.2702391

Yin, S., & Wang, G. (2013). An approach for robust data-driven fault detection with industrial application. *IECON Proceedings (Industrial Electronics Conference)*, 3317–3322. https://doi.org/10.1109/IECON.2013.6699660

Zaidan, A. A., Zaidan, B. B., Qahtan, M. Y., Albahri, O. S., Albahri, A. S., Alaa, M., Jumaah, F. M., Talal, M., Tan, K. L., Shir, W. L., & Lim, C. K. (2018). A survey on communication components for IoT-based technologies in smart homes. *Telecommunication Systems*, *69*(1), 1–25. https://doi.org/10.1007/S11235-018-0430-8/FIGURES/8

Zhao, W., Luo, H., Liu, Q., Ji, H., & Sheng, N. (2022). Incipient Sensor Fault Detection by Directly Monitoring Sliding Window Based Singular Values*. *IFAC-PapersOnLine*, *55*(6), 637–642. https://doi.org/10.1016/J.IFACOL.2022.07.199

Zheng, J., Xu, C., Zhang, Z., & Li, X. (2017). Electric load forecasting in smart grids using Long-Short-Term-Memory based Recurrent Neural Network. *2017 51st Annual Conference on Information Sciences and Systems, CISS 2017*. https://doi.org/10.1109/CISS.2017.7926112