Knowledge-Based Predictive Maintenance for Fleet Management

by

Patrick Killeen

Thesis submitted to the
Faculty of Graduate and Postdoctoral Studies
in partial fulfillment of the requirements
for the Master degree in
Computer Science

School of Electrical Engineering and Computer Science
Faculty of Engineering
University of Ottawa

Abstract

In recent years, advances in information technology have led to an increasing number of devices (or things) being connected to the internet; the resulting data can be used by applications to acquire new knowledge. The Internet of Things (IoT) (a network of computing devices that have the ability to interact with their environment without requiring user interaction) and big data (a field that deals with the exponentially increasing rate of data creation, which is a challenge for the cloud in its current state and for standard data analysis technologies) have become hot topics. With all this data being produced, new applications such as predictive maintenance are possible. One such application is monitoring a fleet of vehicles in real-time to predict their remaining useful life, which could help companies lower their fleet management costs by reducing their fleet's average vehicle downtime. Consensus self-organized models (COSMO) approach is an example of a predictive maintenance system for a fleet of public transport buses, which attempts to diagnose faulty buses that deviate from the rest of the bus fleet. The present work proposes a novel IoT-based architecture for predictive maintenance that consists of three primary nodes: namely, the vehicle node (VN), the server leader node (SLN), and the root node (RN). The VN represents the vehicle and performs lightweight data acquisition, data analytics, and data storage. The VN is connected to the fleet via its wireless internet connection. The SLN is responsible for managing a region of vehicles, and it performs more heavy-duty data storage, fleet-wide analytics, and networking. The RN is the central point of administration for the entire system. It controls the entire fleet and provides the application interface to the fleet system. A minimally viable prototype (MVP) of the proposed architecture was implemented and deployed to a garage of the Société de Transport de l'Outaouais (STO), Gatineau, Canada. The VN in the MVP was implemented using a Raspberry Pi, which acquired sensor data from a STO hybrid bus by reading from a J1939 network, the SLN was implemented using a laptop, and the RN was deployed using *meshcentral.com*. The goal of the MVP was to perform predictive maintenance for the STO to help reduce their fleet management costs.

The present work also proposes a fleet-wide unsupervised dynamic sensor selection algorithm, which attempts to improve the sensor selection performed by the COSMO approach. I named this algorithm the improved consensus self-organized models (ICOSMO) approach. To analyze the performance of ICOSMO, a fleet simulation was implemented. The J1939 data gathered from a STO hybrid bus, which was acquired using the MVP, was used to generate synthetic data to simulate vehicles, faults, and repairs. The deviation detection of the COSMO and ICOSMO approach was applied to the synthetic sensor data. The simulation results were used to compare the performance of the COSMO and ICOSMO approach. Results revealed that in general ICOSMO improved the accuracy of COSMO when COSMO was not performing optimally; that is, in the following situations: a) when the histogram distance chosen by COSMO was a poor choice, b) in an environment with relatively high sensor white noise, and c) when COSMO selected poor sensors. On average ICOSMO only rarely reduced the accuracy of COSMO, which is promising since it suggests deploying ICOSMO as a predictive maintenance system should perform just as well or better than COSMO . More experiments are required to better understand the performance of ICOSMO. The goal is to eventually deploy ICOSMO to the MVP.

Acknowledgements

I would like to thank all the people who helped bring this thesis to fruition: a) Dr. Iluju Kiringa, my thesis supervisor; b) Dr. Tet Yeap, my thesis co-supervisor; and c) Richard Killeen and Suzanne Kettley, who spent many hours proof-reading an earlier version of this report. I would also like to thank the Société de Transport de l'Outaouais, specifically Marc Litalien, Nadine Astresses, Du Lam, and Yannick Carpentier; they were speedy and enthusiastic when responding to requests, and they answered questions with a high degree of expertise. Without their help most of the work done in this thesis would not have been possible.

Table of Contents

List of Tables x

List of Figures xii

1 Introduction 1

1.1 Motivation . . . 1
1.2 Problem - Challenge . . . 2
1.3 Solution Outline . . . 3
1.4 Methodology . . . 4
1.5 Contributions and Limitations . . . 4
1.6 Outline of Thesis . . . 5

Nomenclature 1

2 Background and Related Work 6

2.1 Internet of Things . . . 6
2.1.1 IoT Abstraction Layers . . . 6
2.1.2 Challenges . . . 8
2.1.3 IoT Architectures . . . 9
2.2 Big Data . . . 10
2.3 Gateway . . . 11
2.3.1 Resource Optimization . . . 11
2.4 Cloud Computing . . . 11
2.5 Fog Computing . . . 12
2.6 Edge Computing . . . 12
2.7 IoT-enabling Technologies . . . 13

2.7.1 Hardware . . . 13
2.7.2 Physical and Data-link Layer Protocols . . . 13
2.7.3 Network Layer Protocols . . . 14
2.7.4 Machine to Machine Communication Protocol . . . 14
2.8 Vehicle Maintenance Strategies . . . 15
2.9 Vehicle Diagnostics . . . 16
2.9.1 Controller Area Network . . . 16
2.9.2 On Board Diagnostics - OBD . . . 17
2.9.3 J1939 . . . 17
2.10 Vehicle Prognostics . . . 18
2.10.1 Methodologies . . . 19
2.10.2 Data Characteristics . . . 20
2.10.3 Challenges . . . 21
2.11 Predictive Analytics . . . 22
2.11.1 Predictive Analytics Strategies in Fleet Management . . . 22
2.11.2 Sensor Selection . . . 23
2.12 Statistics . . . 24
2.12.1 Probability Distributions . . . 24
2.12.2 Probability Density Functions . . . 25
2.12.3 Sampling Data . . . 26
2.12.4 Histograms . . . 27
2.12.5 Interquartile Range . . . 28
2.12.6 Entropy . . . 28
2.13 Anomaly Detection . . . 28
2.13.1 Evaluation of Anomaly Detection Algorithms Using ROC Curves . . . 29
2.14 Document Retrieval . . . 34
2.15 Comparisons - Main Approaches and Classification . . . 35
2.15.1 Approach 1 - Estimating p-Values for Deviation Detection . . . 35
2.15.2 Approach 2 - A Field Test With Self-organized Modeling for Knowledge Discovery in a Fleet of City Buses . . . 37
2.15.3 Approach 3 - Self-Organizing Maps for Automatic Fault Detection in a Vehicle Cooling System . . . 38
2.15.4 Approach 4 - Consensus Self-organized Models Approach — COSMO 40

2.15.5 Approach 5 - MineFleet . . . 47
2.15.6 Approach 6 Cloud-based Driver Monitoring and Vehicle Diagnostic with OBD2 Telematics . . . 48
2.15.7 Approach 7 - An IoT Gateway Centric Architecture to Provide Novel M2M Services . . . 50
2.15.8 Approach 8 - Connected Vehicle Diagnostics . . . 51
2.15.9 Approach 9 - Military Vehicle Diagnostics . . . 53
2.15.10 Approach 10 - An IoT Gateway Middleware for Interoperability in SDN Managed Internet of Things . . . 54
2.15.11 System Requirement . . . 55

3 My Approach - IoT-based Architecture of a Bus Fleet Management System 58

3.1 Meeting IoT and Fleet Management System Requirements . . . 58
3.2 Architecture Layers . . . 60
3.3 Fleet Remote Administration . . . 61
3.4 Vehicle Node . . . 61
3.4.1 Security . . . 62
3.4.2 Data Acquisition . . . 62
3.4.3 Data Analytics . . . 62
3.4.4 Network Connection . . . 63
3.5 Server Leader Node . . . 63
3.5.1 Security . . . 63
3.5.2 Data Acquisition . . . 64
3.5.3 Data Analytics . . . 65
3.5.4 Network Connection . . . 65
3.6 Root Node . . . 66
3.6.1 Security . . . 67
3.6.2 Fleet System Interface . . . 67

4 Fleet-wide Predictive Maintenance Using Knowledge-base 69

4.1 Motivation . . . 69
4.2 Abbreviations . . . 69
4.3 Definitions . . . 70

4.4 Assumptions . . . 71
4.5 Sensor Set Definitions . . . 72
4.5.1 Sensor Set Comparison . . . 73
4.5.2 Sensor Set Properties . . . 73
4.6 Methodology . . . 74
4.7 Improving Deviation Detection . . . 75
4.7.1 Optimal Sensor Set Analysis . . . 76
4.7.2 Finding Sensors Involved in a Fault . . . 76
4.7.3 Adjusting Sensor Contribution and Potential Contribution . . . 80
4.7.4 Dynamic Sensor Selection . . . 81
4.8 Knowledge Discovery . . . 83
4.8.1 Faulty Sensor Detection . . . 84
4.8.2 Suggesting Sensor Installation . . . 84

5 Implementation and Results **85**
5.1 Predictive Maintenance Architecture System Prototype . . . 85
5.1.1 STO Requirements Gathering . . . 85
5.1.2 Gateway Implementation . . . 85
5.1.3 STO Garage Installation . . . 87
5.1.4 Data Acquisition . . . 89
5.2 Sensor Data Analysis . . . 90
5.2.1 Data Format . . . 90
5.2.2 J1939 Specification Document Parsing . . . 90
5.2.3 J1939 Packet Analysis . . . 91
5.2.4 Data Exploration . . . 91
5.3 Fleet Simulation Implementation . . . 94
5.3.1 Problem Description . . . 95
5.3.2 Assumptions . . . 96
5.3.3 Definitions . . . 97
5.3.4 Random Number Generation . . . 97
5.3.5 Configuration Phase . . . 98
5.3.6 Generation Phase . . . 100
5.3.7 Analysis Phase . . . 120

5.4 STO Sensor Data Sampling Analysis . . . 130
5.5 Synthetic Sensor Data Generation Analysis . . . 131
5.5.1 Sensor Behavior Parameter Analysis . . . 131
5.5.2 Stability Analysis . . . 132
5.6 Fleet Simulation Result Analysis . . . 133
5.6.1 Goal . . . 133
5.6.2 Methodology . . . 133
5.6.3 Simulation Configuration . . . 134
5.6.4 Histogram Distance Analysis . . . 136
5.6.5 Analyzing ICOSMO Parameter Effects . . . 138
5.6.6 ICOSMO vs. COSMO . . . 146
5.6.7 Summary . . . 151

6 Conclusion **153**

APPENDICES **155**

A Results and Diagrams **156**

B Configuration and Source Code **165**
B.1 Hardware . . . 165
B.1.1 Raspberry Pi . . . 165
B.1.2 Fona 800 . . . 169
B.1.3 UPS PIco . . . 171
B.1.4 Copperhill ECU . . . 172
B.2 Configuration for the Raspberry Pi for the Gateway . . . 172
B.2.1 Configuration for the Fona 800 . . . 173
B.2.2 Configuration for the UPS PIco . . . 175
B.2.3 Reading from the J1939 Network Using Copperhill ECU . . . 178
B.3 Configuring the STO Laptop . . . 179
B.3.1 SLN Data Acquisition Details . . . 181
B.4 J1939 and Java . . . 182
B.4.1 Parsing J1939 Specification Document using Java . . . 182
B.4.2 J1939 Packet Analysis using Java . . . 187

B.5 Fleet Simulation Java Code 189
B.5.1 Noise Generation 189
B.5.2 Configuration Phase 190
B.5.3 Generation Phase - COSMO 191
B.5.4 Analysis Phase 192
B.5.5 Analysis Phase - COSMO 193
B.5.6 Analysis Phase - Output 193
B.6 Creating ROC Curves Using the R Programming Language 194

References **197**

List of Tables

2.1 Computing a ROC curve example: labeled samples and outlier score for 16 samples, where "F" = False and "T" = True. 31

2.2 Computing a ROC curve example: evaluation results for threshold = 0.45, where "F" = False, "T" = True, "FN" = False Negative, "TN" = True Negative, "TP" = True Positive, and "FP" = False Positive. 32

2.3 Document retrieval: object label assignment based on relevant or irrelevant objects retrieved. 35

2.4 Comparison of the Approaches Found in Section 2.15, using requirements defined in section 2.15.11. Where, "y" = yes (the requirement is met), "n" = No (the requirement is not met), "p" = partial (the requirement is somewhat met), "u" = unclear (the requirement may or may not have been met, it was not clear), and "-" = not applicable (the requirement is not relevant or applicable). 57

4.1 Information retrieval algorithm possible predictions. 79

5.1 Statistics revealing the presence of noise for an anonymous sensor. 92

5.2 Fault-involved sensors' behavior for a run-away engine fan FaultDescription. 103

5.3 Notations used to denote the inner elements of m, where $m \in M$. (*) denotes a parameter used when $m_{x,D}$ denotes a normal distribution, and (**) denotes, when $m_{x,D}$, a uniform distribution. 106

5.4 Notation for the SensorBehavior class attributes. 106

5.5 The mean estimated probabilities, for 61 sensors, that their sample mean fall within confidence intervals, where i denotes the confidence interval as follows: $P(\mu_{-i} < \overline{X} < \mu_i)$ and f_p denotes the fraction of sensor readings filtered (1 - f_p denotes the fraction of readings sampled). 131

5.6 Performance of the COSMO approach when using various histogram distances for H_{1-3}, where Dist. = Distance and AUC = Area Under the Curve. 141

B.1 Complete list of hardware components used to build the gateway. 166

B.2 STO vehicle node (Raspberry Pi) test cases. 167

B.3 STO server leader node (laptop) test cases. 168
B.4 Fona 800 bottom breakout pins, top-down order with respect to the orientation of the Fona 800 in figure B.1. 170
B.5 Fona 800 useful serial commands for debugging. 174
B.6 J1939 Specification Document, consistent sensor definition field formats. . 182
B.7 J1939 Specification Document, possible sensor definition formats for the `SPN Position in PGN` field. Note, ESV is an abbreviation for encoded sensor value. 183
B.8 J1939 Specification Document, possible sensor definition formats for the `SPN Length` field. 184
B.9 J1939 Specification Document, possible sensor definition formats for the `Resolution` field. 185
B.10 J1939 Specification Document, possible sensor definition formats for the `Offset` field. 186
B.11 J1939 Specification Document, possible sensor definition formats for the `Data Range` field. 186
B.12 J1939 Specification Document, possible sensor definition formats for the `Operational Range` field. (*) denotes: x is the minimum and y is the maximum operational range. (**) denotes x is the minimum and y is the maximum operational range, where u is a string representing the units. . . 187

List of Figures

2.1 Probability of a random variable X of falling in the area between a and b under the curve f(x), inspired from [67] 25

2.2 ROC curve true positive rate vs. false positive rate, inspired by [32] 33

2.3 ROC curve point interpolation example, where the red X indicates which points are removed, and the orange line represents the interpolation from the point (0.5,0.6) to the point (0.8,0.7). 34

3.1 Predictive maintenance fleet management system architecture overview diagram. ECU = Electronic Control Unit, VN = Vehicle Node, SLN = Server Leader Node, RN = Root Node, WAN = Wide Area Network, CAN = Controller Area Network. Wireless Access Point includes Wi-Fi, LTE, GSM, Bluetooth, etc. 59

3.2 SysML [38] diagram of vehicle node. ECU = Electronic Control Unit. . . . 64

3.3 SysML [38] diagram of server leader node. 66

3.4 SysML [38] diagram of root node. 68

4.1 Sensor set ven diagram, where the set I is the yellow area, U is the blue area, C is the white circle, and E is everything outside the white circle. . 74

4.2 All cases of possible optimal set O's configurations. 75

4.3 Where P are all the fault-involved sensors, $\overline{P}$ are all the non-fault-involved sensors, and X are the sensors estimated by the BBIRA. 79

5.1 Hardware SysML [38]-style diagram of gateway. 86

5.2 MVP gateway hardware box design: i) original metal box design; ii) plastic box hardware migration; iii) MVP's waterproof gateway box. 87

5.3 Simple diagram of the MVP architecture deployed at the STO garage. . . . 88

5.4 Anonymous sensor's behavior over time, with presence of outliers. 93

5.5 Example of a constant synthetic sensor data stream with 8 unique noise samples (virtually cardinality = 1). 94

5.6 Anonymous liquid percent remaining graph. 95

5.7 UML class diagram of the core classes used in this simulation. 96
5.8 UML class diagram of the noise class, which is used for generating random numbers. 98
5.9 Fault generation model UML class diagram. 104
5.10 Fault generation model communication diagram, where $AFIS$ = active fault-involved sensors. 105
5.11 Sensor data generation model UML diagram. 107
5.12 Sensor data generation model communication diagram, when generating synthetic data. 108
5.13 Sensor data generation model communication diagram when processing faults and repairs. 110
5.14 COSMO Deviation Detection Model SensorStatusEvent UML Class Diagram 111
5.15 COSMO deviation detection model, simplified UML class diagram. 112
5.16 History UML class diagram. The associations represented by dashed-lines are Java HashMaps. For example, the *keys* association is of the form `HashMap<K,HashMap<TimerEvent,List<Event>>>`. 117
5.17 History communication diagram. Note that figure 5.18 illustrates the additional steps performed in the FaultHistory class. 118
5.18 FaultHistory communication diagram. 119
5.19 ICOSMO Implementation UML Class Diagram. 124
5.20 Comparison of histogram distances, when used in COSMO, using average AUC. 137
5.21 Comparison of histogram distances, when used in COSMO, using average AUC. High sensor noise and large simulation 138
5.22 Comparison of histogram distances, when used in COSMO, using standard deviation of AUC. High sensor noise and large simulation. 139
5.23 Experiment set 1. Average AUC performance, Type A experiment using H_1, desired precision 5%, increasing estimated fault-involved sensor and maximum added sensors parameters. Better performing histogram distances (Bhattacharyya, Hellinger, and Matusita) are on the left and the poor histogram distances (others) are on the right. 140
5.24 Experiment set 2. Average AUC performance, Type A experiment using H_1, desired recall is increased from 10% to 85%. Better performing histogram distances (Bhattacharyya, Hellinger, and Matusita) are on the left and the poor histogram distances (others) are on right. 143
5.25 Experiment set 3. Average AUC performance, Type A experiment using H_2, where *General Performance Trend* is the general behavior of all the histogram distances other than the City Block and Fidelity distances. . . . 144

5.26 Experiment set 3. Type A experiment set on H_2. The average area under the curve (y-axis) for all histogram distances with respect to the potential contribution increase modifier (x-axis), with 70% desired precision. 145

5.27 Experiment set 6. , ICOSMO vs. COSMO average AUC performance, Type B experiment. 147

5.28 Experiment set 7. Example of ICOSMO improving the accuracy (average AUC) of poor COSMO sensor selection choices. 148

5.29 Experiment set 7. Example of ICOSMO improving the accuracy (average AUC) of poor histogram choices. 149

A.1 Example of a probability density function for a uniform distribution with minimum = 0 and maximum = 1. 157

A.2 Example of a probability density function for a normal distribution with mean = 0 and standard deviation = 1. 157

A.3 Example of a probability density function for a skewed/beta distribution, where shape1 = 3, and shape2 = 7. 158

A.4 Example of a probability density function for an exponential distribution. . 158

A.5 Example of a probability density function for a Bimodal distribution. It is composed of 2 normal distributions. The first normal distribution nas mean = 0 and standard deviation = 1, and the second normal distribution has mean = 5 and standard deviation = 1.5. 159

A.6 Experiment set 8. Comparing the average AUC of COSMO and ICOSMO with high and low sensor white noise. 159

A.7 Stability analysis, experiment 1: stability vs. Δp_x values when generating synthetic data using Equation 5.1, using uniformly distributed white noise. The original dataset is illustrated in Figure A.9. 160

A.8 Stability analysis, experiment 2: stability vs. Δp_x values when generating synthetic data using Equation 5.1, using normally distributed white noise. The original dataset is illustrated in Figure A.9 160

A.9 Histogram of anonymous STO sensor data with 215,348 samples. 161

A.10 Analysing the effects of generating a synthetic dataset of 1000 samples (using Equation 5.1) with $\alpha = \{0.5, 1, 2\}$ and $p = \{0, 0.4, 0.7\}$. Note that the y-axis for each histogram is the normalized frequency. 162

A.11 Confidence levels for sensor data sampling strategies. 163

A.12 ICOSMO AUC error when varying threshold decimal precision. 163

A.13 COSMO AUC error when varying threshold decimal precision. 164

B.1 Fona 800 board. 170

B.2 UPS PIco Board, with pin labeling from [78], where output pins that are used in the gateway are highlighted. 171
B.3 Copperhill J1939 ECU diagram, where interfaces that are used in the gateway are labeled. 172
B.4 Raspberry Pi configuration tool console user interface. 176
B.5 Example output of the Raspberry Pi's I^2C devices. 177
B.6 Example Output of UPS PIco status using `pico_status.py` Python script. 177
B.7 Example of extracting a subset of bits from a payload. 189
B.8 Bit operations required to extract bits 16 to 24, a subset of bits from a payload. 190

Chapter 1

Introduction

Internet of Things (IoT) is a new paradigm that is growing quickly. By 2020 many believe billions of devices will be connected to the internet [10][72]. With all these devices connected to the internet, big data becomes more prevalent and can be used by applications to acquire new knowledge. Modern data analytic algorithms struggle to process the massive amounts of data produced by these IoT devices. Some of the IoT applications include smart farming, smart transport, smart health, smart cities, smart homes, and smart grids [10][20][21][23][61][31][11]. Predictive maintenance, an example of smart transportation, attempts to predict the health of equipment using machine learning and/or artificial intelligence. Common prediction measures for equipment include remaining useful life and health status. However, predicting vehicle faults is not a trivial task for several reasons, a) good sensor selection is required; b) labeled fault data is generally difficult to obtain compared to normal behavior data, since it is not always financially viable to break equipment to gather fault data; and c) it is not always immediately obvious which machine learning or artificial intelligence algorithm is the best choice for a predictive analytics task.

The present work proposes an IoT-based fleet management system architecture for performing predictive maintenance and proposes a fleet-wide data analytics algorithm designed to run on the proposed architecture. A live prototype of the architecture is deployed with the help of a public transport bus company named Société de Transport de l'Outaouais (STO), and fleet simulations are run to evaluate the performance of the proposed fleet-wide data analytics algorithm. An earlier version of this work can be found in [51], and a rule-based predictive maintenance framework that is based on the same predictive maintenance STO project can be found in [16].

1.1 Motivation

In the past, large companies with fleets of expensive vehicles (for example, airline companies), could afford designing, implementing, testing, and deploying predictive maintenance systems to avoid unscheduled downtime and reduce fleet management costs. However, smaller companies with relatively less expensive equipment could not benefit from such

systems, since often it was more cost effective to let their equipment break down and react to the fault instead. That is, rather than investing into an expensive system to reduce unscheduled repairs by predicting faults of less expensive equipment, allowing the faults to occur or preemptively replacing components was more financially feasible. However, in recent years, technological advances have resulted in cheaper and smaller hardware (such as sensors and processing power), and communication costs have also become cheaper. As a result, predictive maintenance system deployment has become financially possible for smaller fleet companies with less expensive equipment.

Most heavy-duty machinery is equipped with a Controller Area Network (CAN), which allows equipment subsystems to communicate with each other to enable the equipment's operation and self-diagnostics. An important category of information found in CAN traffic is sensor data. Most heavy-duty machinery is not connected to the internet, and as a result this sensor data remains unused. Therefore, there is an opportunity to perform predictive analytics on heavy duty machinery by acquiring and analyzing all this under-utilized sensor data.

A recent trend in the vehicle industry for performing predictive analytics on this unused sensor data, according to [104], involves uploading the sensor data to a server for offline fault analysis. That is, upon discovering a fault, once the vehicle is brought into the garage, the sensor data associated with a fault is uploaded to a server. Each vehicle's sensor data is then individually and manually analyzed by an analyst; however, as fleet size increases, this approach becomes infeasible [104]. It would be difficult for an analyst to analyze the data of an entire fleet and find fleet-wide (or cross-fleet) trends. An automated fleet-wide predictive maintenance system is therefore required. The mechanics could then analyze the knowledge provided to them by the system instead of the raw data, which could help the analysts discover new faults and trends in the fleet's data that otherwise would not have been discovered.

STO is an example of a public transport company, located in Gatineau, Québec, Canada, that has a fleet of buses equipped with a CAN-based system, J1939, but do not make use of the sensor data. The STO only use J1939 data for diagnostic purposes when performing maintenance on buses using specialized equipment. There is therefore an opportunity to help them with their fleet management by deploying a predictive maintenance system to assist them. By accessing the sensor data in real time (or pseudo real time) over the internet, on-board and fleet-wide predictive analytics can then be applied to discover new knowledge. By using this knowledge, in addition to performing their scheduled preventive maintenance, the STO could adapt their fleet management strategies to reduce unscheduled downtime, the average vehicle garage-time for their fleet, and other maintenance related costs.

1.2 Problem - Challenge

The predictive maintenance solution should be general enough to be applied to many other architectures, contexts, and business problems, and it should be automated in the

sense that it should not require human intervention to maintain the system. To perform predictive maintenance, the sensor data from the fleet vehicles must be acquired, and this data is either analyzed in real-time, offline, or using a combination of both. The first challenge is to install an IoT **gateway**, a small computer, onto the STO's vehicles. The gateway should have the ability to read J1939 sensor data and to connect to the internet via a wireless connection. Once the gateway is installed, the sensor data must be read and accessible by the system for fleet-wide analysis. The data cannot be blindly sent over the wireless network from the vehicle to a cloud server due to the following resource constraints: power restrictions, computational limitations, low communication bandwidth, and high communication monetary costs. Therefore, one of the challenges is **data acquisition**; that is, gathering all the necessary sensor data from each vehicle, sending them over a wireless network, and storing them in a central location to enable fleet-wide analysis. Since the vehicles are constantly moving, the internet connection will need to be configured via wireless telemetry such as Wi-Fi or Long-term evolution (LTE/4g).

Once the data has been acquired, the next challenge is performing **data analysis**. Some data analysis can be done on-board the vehicle (on the gateway) and other analysis can be done in the cloud. This thesis focuses mostly on fleet-wide analysis performed in the cloud. Furthermore, all the sensor data streams cannot be naively included into the machine learning models due to machine learning performance limitations. Thus meaningful sensor data must be chosen to achieve accurate predictions and minimize information loss, which can be achieved using **sensor selection**. To avoid sending all the sensor data over the wireless network, on-board aggregation strategies can be used; for example, modeling the sensor signals using histograms. It is not efficient to create a machine model for each individual vehicle; instead, designing a general model that can be applied to multiple vehicles should be the approach taken [17]. Furthermore, the predictive analytics model should be able to evolve through time and be applicable to many different types of vehicles.

1.3 Solution Outline

I propose an IoT-based fleet management architecture, which supports data acquisition, on-board data analytics, and fleet-wide analytics. Each vehicle is equipped with a gateway (the core processing unit on-board the vehicle). The gateway enables J1939 sensor data acquisition, on-board data analytics, and acts as the bridge between the J1939/CAN bus and the internet. The server leader nodes are responsible for managing vehicles in a geographical area and performing fleet-wide data analytics. The server leader nodes are all controlled by a central point of administration of the entire fleet system, the root node. I also propose a fleet-wide unsupervised dynamic sensor selection algorithm, ICOSMO, which attempts to improve the accuracy of the deviation detection of the COSMO approach (mentioned in [82]) by dynamically ranking and selecting sensors.

1.4 Methodology

To improve the STO's fleet management with a predictive maintenance system I first gathered requirements for such a system. I did this by conducting many interviews with STO mechanics and fleet managers to gather the requirements. While gathering the requirements, I conducted IoT and predictive maintenance research in parallel. Once I gathered the requirements and performed a sufficient amount of research, I designed ICOSMO and an IoT-based fleet management architecture using UML and SysML (defined in [38]), which I used to formalize the design description of the architecture. To deploy a prototype of the proposed architecture at the STO's garage, I organized two additional meetings. The first meeting was to check whether data acquisition was possible with the prototype. If the prototype failed to acquire data, the project would have halted. However, the first meeting confirmed the J1939 data acquisition was indeed possible using the prototype. The second meeting was scheduled to install and deploy the prototype. With the prototype in place, data was remotely gathered for offline analysis. After analyzing the data, it was time to design a simulation that could simulate a fleet of vehicles using the acquired data. The simulation would be used to evaluate the performance of the COSMO and ICOSMO approach, by evaluating their fault detection performance.

1.5 Contributions and Limitations

Contributions:

- An extensive survey of related work on anomaly detection for predictive maintenance. This survey analyzes the advantages and disadvantages of the related work. It also compares the related work to each other and to the work done in this thesis, using the fleet management and IoT requirements gathered in the literature review of this thesis.
- An IoT-based layered architecture for fleet management that distinguishes cloud-based components from edge-based ones, that consists of three primary nodes: namely, the vehicle node (VN), the server leader node (SLN), and the root node (RN). The VN represents the vehicle and performs lightweight data acquisition, data analytics, and data storage. The SLN is responsible for managing a region of vehicles, and it performs more heavy-duty data storage, fleet-wide analytics, and networking. The RN is the central point of administration for the entire system.
- A fleet-wide unsupervised dynamic sensor selection algorithm, ICOSMO, that attempts to improve the deviation detection performance of the COSMO approach. ICOSMO estimates the deviation detection contribution of each sensor. Stale sensors (those that fail to deviate in response to a fault) are removed from the selected features, and candidate sensors (those that would most likely benefit the deviation detection performance) are added to the selected features.

- An implementation and deployment of a STO prototype that gathers J1939 sensor data used to generate a bus fleet simulation. A single STO bus has a gateway installed, which sends the sensor data to a local server in the garage. The sensor data is remotely accessible for offline analysis via a LTE internet connection configured for the server in the garage.
- An analysis of the performance results of the fleet simulation used to evaluate the ICOSMO and COSMO approaches. The simulation generates synthetic data to simulate vehicles, faults, and repairs. The deviation detection of the ICOSMO and COSMO approach was applied to the synthetic sensor data. Various parameters of ICOSMO were manipulated to gather insight on their effect on the performance results.

Prototype Limitations:

a) Only one vehicle had a gateway installed on it.

b) The experimental fleet simulations, which are used to evaluate the ICOSMO and COSMO approach, would have benefited from including more sources of real data than those from the single real gateway.

c) The data was not labeled so normal and fault data were not explicitly defined.

d) ICOSMO assumes that access to a vehicle service record database (VSRDB) is available, but in the case of the STO no such database was available.

e) ICOSMO assumes an information retrieval algorithm exists to estimate fault-involved sensors using VSRDB repair records as queries, but no such algorithm exists.

f) Due to the limitations d) and e), deploying ICOSMO onto an STO bus should have been accompanied by the development of a VSRDB. However, such an endeavour was out of the scope of this thesis.

1.6 Outline of Thesis

This thesis is structured as follows: chapter 2 provides some background and a literature review of concepts that are related to IoT and predictive maintenance, and it also compares related work to the work proposed in this thesis; chapter 3 proposes an IoT-based architecture for fleet management; chapter 4 proposes a fleet-wide unsupervised dynamic sensor selection algorithm; chapter 5 discusses the implementation details and results (specifically, the implementation and deployment details of the STO prototype, an exploration and analysis of the data acquired from the prototype, the implementation details of the bus fleet simulation, and the analysis of the performance results of the simulation to evaluate the ICOSMO and COSMO approaches); and section 6 concludes this work and discusses future work.

Chapter 2

Background and Related Work

2.1 Internet of Things

Internet of Things (IoT) connects many heterogeneous devices [10][20][72][23][52], and these devices can sense and interact with the environment around them. A service provider company named Libelium is working on providing global wireless sensors network coverage, LoRaWAN, to support global IoT deployment. They began covering Asia-Pacific including China and India [58]. Some of the key requirements to IoT is low-latency and low-bandwidth usage [23].

2.1.1 IoT Abstraction Layers

A literature review of [20][21][61][52][73] reveals that IoT can be split into five abstraction layers: sensing, network, storage, learning, and application. These abstraction layers are required to seamlessly integrate many heterogeneous IoT devices together, which enables them to communicate with each other, and are required to provide clients with access and control to IoT devices. These abstraction layers also hide many low level implementation details that are unnecessary for clients. For example, clients do not need to know which lower level sensor network protocol is used [23].

2.1.1.1 Sensing Layer

The sensing layer gathers data from the environment and interacts with it, using sensors and actuators. Data acquisition and sensor selection are examples of functionality that can be found in this layer.

2.1.1.2 Network Layer

The network layer connects lower level nodes to the cloud/fog [10][72]. It is in charge of managing all the interplay between the edge nodes, fog nodes, and cloud nodes. Each of

these nodes has a variety of resource constraint requirements, and this layer is responsible for meeting these requirements. Some machines do not have enough resources to implement full-fledged networking protocols [7]. Instead they must use machine-to-machine (M2M) communication protocols, which are more lightweight on the client side (see section 2.7.4 for more details). Another example of functionality implemented in this layer is that some sensor networks or IoT devices do not have enough resources to implement network security mechanisms. By connecting a service provider to such devices, a layer security can be applied over these insecure devices [7].

2.1.1.3 Storage Layer

The storage layer stores sensor data, aggregations, and other types of data. To meet IoT storage requirements, this layer can store data using a variety of data storage strategies, and it can choose the best location to store the data. For example, sensor data may best be stored on a gateway (for more information see section 2.3) in the fog (for more information see section 2.5), aggregations of the data in the cloud, and knowledge discoveries in a client application. The storage layer should always include the original value of a data point when storing data, to reduce the risk of losing data [7]. When storing sensor data using a relational database, [7] suggests creating a table for each of the sensors to reduce information loss and other redundant data.

2.1.1.4 Learning Layer

The learning layer performs data analytics on stored sensor data for knowledge discovery. This layer is responsible for managing where the intelligence of the system is performed (in the edge, fog, or cloud for example). This layer attempts to optimize the location of the data analytics to best meet the target system's requirements. For example, a single-vehicle machine learning model should be run on a gateway, and fleet-wide analytics should be run on the cloud.

2.1.1.5 Application Layer

The application layer provides the interface to the IoT system by providing clients with access (and control) to IoT devices. The application layer should represent data in a user friendly way [7]. Furthermore, this layer should separate the IoT data database server from the application web servers for security reasons [7].

2.1.1.6 IoT Layer Interaction Examples

Network and Sensing Layer Interaction Example

Time-stamping sensor readings can be handled in this layer. Suppose a gateway is sampling sensor readings and sending them (or aggregations of them) to a cloud server.

[7] proposes two options for time-stamping, either the time-stamping is performed on the gateway or in the cloud. Each option has its advantages and disadvantages. Option a) is the best option if the accuracy of the timestamps is critical to the system, since the timestamps are created closer to the data source and are not affected by the latency between the cloud server and the gateway. In the case of option b), the timestamps will be proportionally inaccurate to the amount of latency between the gateway and cloud server. However, option a) requires more resources from the gateway while option b) delegates the time-stamping to the cloud server, which has more resources.

Network and Learning Layer Interaction Example

Another example mentioned by [7] is as follows: when sorting the moving average (or some other aggregate) of a sensor value, there are 2 options; a) to perform the average calculation on the gateway (close to the data source), or b) calculate the average on the cloud server. Option a) is favorable when network bandwidth is low, since aggregates will be sent to the server, and option b) is favorable when the gateway has low computational resources and its bandwidth is not restrictive.

2.1.2 Challenges

According to [72], there are many challenges with IoT, which include the following: a) determining who owns data. For example, a patient with smart pacemaker has his or her data uploaded to the cloud via a service provider, and the data is analyzed by a third party. Deciding whether the data belong to the patient, the manufactures of the device, or the cloud analytics provider is not trivial. The same question can be asked about knowledge obtained from data mining the pacemaker data; b) determining which party can monetize the data. When money is made using the patient's heartbeat data, deciding whether the patient deserve to be payed is also not trivial; and c) IoT is missing a universal standard. As IoT becomes more widespread, the heterogeneous devices will need to be able to communicate with each other. Therefore, interoperability between all these devices will be a key challenge.

2.1.2.1 Data Stream Challenges

IoT is naturally tied to data stream processing. This section summarizes the issues and challenges, mentioned in [101], that may occur when processing multiple data streams. The challenges and issues are as follows: a) typically, a data acquisition node near the data sources, a gateway for example, has a limited amount resources (such as battery, storage, and computation power); b) one cannot treat multiple data streams as one stream with many attributes, since the attributes may have different distributions in time, sampling rates, and data processing models; c) it is generally required to have the ability compare streams on the edge, fog, and cloud; d) multiple heterogeneous data streams are more difficult to process compared to multiple homogeneous data streams. At times there are many different sources in time and data distribution, which means each heterogeneous data stream must be processed differently; e) stream synchronization; f) the trade-off between

accuracy and computational power is another challenge. It may not be feasible for an IoT device to sample at the output rate of a sensor, so there will be information lost. A similar trade-off also exists between accuracy and communication costs; g) identifying the data streams is a challenge since they may be inherently anonymous; h) keeping the data streams private may hinder the accuracy of the data collected; and i) covert channels must also be considered, that is, the privacy of knowledge found/inferred from data streams.

2.1.3 IoT Architectures

There are different types of IoT architectures, where each are designed to meet a different subset of IoT requirements. [15] and [70] mention a variety of different IoT middleware architectures. The **service-based IoT middleware** architecture provides access to deployed IoT devices as a service to users. This type of IoT middleware focuses on facilitating IoT device deployment. The **cloud-based IoT middleware** architecture is more strict when it comes to IoT device deployment, but it enables better data collection, storage, visualization, and cloud-based data analytics. The cloud in this type of architecture provides a predefined API for users to access the system. The **actor-based IoT middleware** architecture allows resource-constrained devices to access the sensors and actuator via the cloud, which means this architecture can therefore be deployed to the fog.

2.1.3.1 Architectures Layer Interfaces

Designing the interface between the logical layers of the IoT architectures is important [73]. The *Fog-to-Cloud* interface is required to provide services to users from the application layer. That is, applications connected to the cloud need access to an API that lets them access fog-connected IoT devices. Depending on system requirements, storage and data processing can either be performed in the cloud or the fog. The *Fog-to-Fog* interface controls the fog network management. This interface allows fog nodes to communicate with each other. For example, an IoT gateway installed on a vehicle (in the fog) can communicate to another vehicle via a cellular data plan connection. The *Fog-to-Fog* interface in this example is the cellular data plan service provider. The *Fog-to-Things* interface (which is the interface that allows fog devices to communicate with edge devices (or things), such as sensors) enables the *Cloud-to-Thing* interface to exist. The *Cloud-to-Thing* interface is the interface that allows cloud-connected applications to access sensors and actuators. These interfaces are important, since together they act as the bridge between the end user and the IoT data.

2.1.3.2 Challenges

[73] mentions a few key challenges when designing IoT architectures, which include a) coordinating services with the fog nodes and managing energy; b) scalability of the system; c) node mobility in terms of persistent connectivity and handover. The goal is to have the least amount of downtime, and delegate/offload services across the network when links fail;

and d) security, which includes data integrity and authentication at all level of the IoT architecture.

2.2 Big Data

The rate of data creation is increasing exponentially [12][40], which is a problem with limited processing power [5][25]. The cloud in its current state will have difficulty handling all this data [10][23][52], that is, standard technologies for processing data will struggle [72]. According to [8] and [52], the five V's of big data are as follows: *volume*, *veracity*, *velocity*, *variety*, and *value*. Tera bytes of data are produced each day by thousands of vehicles, which is an example of big data in the vehicle industry [45].

According to [42], big data systems have to deal with massive numbers of heterogeneous datasets, distributed data sources, scalability, and the ability to mine big datasets in real-time or pseudo real-time. They must also provide the ability to visualize the data and optimize predictions using the data. Hadoop is a popular tool for dealing with big data analytics. Hadoop's MapReduce is tool that is designed to handle big datasets. Hadoop is an open source framework that consists of three main components: Hadoop distributed file system (HDFS), MapReduce, and YARN [23][7]. Traditional relational database management systems fail to meet the requirements for big datasets [23]. Big data has little structure, while relational database management systems (RDBMSs) support only structured data, and therefore RDBMS fails to scale as datasets become large (it is expensive to upgrade hardware to scale the RDBMSs).

Big data can be defined from multiple point of views, which include the following: a) from an attributive point of view (the five Vs); b) from a comparative point of view (datasets that are too large to be handled by standard data management tools); and c) from an architectural point of view, where the following make it difficult to analyze big data: the distribution, heterogeneity, number of data sources, volume, and speed. In IoT big data also includes the number of devices deployed.

The history of data and how it scaled over the years is as follows:

1. Megabyte to Gigabyte: 1970 and 1980s, they created database systems to manage the data
2. Gigabyte to Terabyte: In the late 1980s, they researched parallel databases to manage the data
3. Terabyte to Petabyte: In the late 1990s, MapReduce and Google File System were introduced by Google, to manage the unstructured data
4. Petabyte to Exabyte: Current era, no robust big data solution for all the data has been proposed. Research in big data is expanding.

2.3 Gateway

A literature review of [7], [10], [15], [52], and [21] reveals that gateways are fog nodes that can be found in the sensing, storage, learning, and network IoT layers. Gateways gather heterogeneous sensor data streams, aggregate the data, perform lightweight data analytics, and store the data temporarily before sending it (and/or analytical discoveries) to a cloud database server. Raspberry Pi and Arduino are examples of gateway hardware solutions. Gateways also connect edge nodes to the internet and enable edge device management. In other words, gateways implement the *Fog-to-Thing* IoT architecture interface (see section 2.1.3.1 for more details), since it is a fog node connecting to things (a wireless sensor network, for example). Since the gateway has an internet connection (via Wi-Fi, LTE, or Ethernet, for example), it implements the *Cloud-to-Thing* IoT architecture interface by providing the bridge between the internet and the things. In other words, gateways abstract their underlying sensor networks to provide an edge device API for higher level nodes in the software stack [21]. This API can then be secured to add a layer of security over the insecure sensor networks [7].

2.3.1 Resource Optimization

According to [92] and [52], gateways are well suited for data locality (and therefore increase availability), since they lie near the data source and have the computational resources required to perform lightweight stream analysis and aggregations. Only aggregates of the sensor data streams should be sent to the cloud if possible [10]. To further optimize resource utilization, according to [23], compressing the data from multiple sensor data streams can optimize data storage, network transmission, and analytics. However, IoT data compression is different than the standard data compression techniques. For example, one strategy involves removing redundant information from the streams and another strategy involves minimizing information loss.

Furthermore, when gateways store data locally, they should use an embedded database, such as SQLite, instead of a local database management system (DBMS), since otherwise it would create unnecessary latency [86][52].

2.4 Cloud Computing

The cloud offers three main services: Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure (hardware) as a Service (IaaS) [72][23][73]. According to [23][73],

- SaaS provides applications to users, such as a database instance hosted by Microsoft Azure.
- PaaS provides a middleware for applications so that the infrastructure for a new application does not have to be built from scratch.

- IaaS provides virtual machines, networking, storage and computing.

Cloud computing provides the platform, resources, and network for various applications, which are not easily accessible by IoT devices. The cloud can therefore be used to abstract the IoT devices, which are heterogeneous, resource-constrained, and insecure by nature [23]. The cloud platform as a whole will have difficulty dealing with the increasing number of IoT devices being deployed daily, and will have trouble supporting the communication between these devices [73]. Due to the real-time nature of IoT application, the cloud will also have trouble meeting the real-time requirement of IoT [93].

Tools and platforms that can help support the cloud with IoT applications follow: a) a distributed database management system (DDBMS), which is a distributed system designed to manage data like a traditional DBMS, but with high availability and low latency requirements [23], and b) Microsoft Azure, Google Cloud platform, Amazon web services, and IBM cloud, which are examples of the leading cloud service providers [73].

2.5 Fog Computing

Cisco was the first that introduced the notion of fog computing [93]. The fog provides the ability to reduce network bottlenecks, improve security, and handle scalability issues [73]. The fog is a resource-constrained cloud near the edge of the network, which supports low latency and lightweight computing [15]. It performs computations at the edge of the network [73] and in close proximity to its mobile users [93]. It meets some IoT requirements that are not satisfied by the cloud [23], such as real-time computations and local content [73]. The fog also provides the cloud with an interface to the lower IoT software layers [10]. The fog and cloud complement each other, but the fog is not going to replace the cloud [73][10]. Some of the applications of fog computing are as follows: gaming, video conferencing, geographically distributed applications (for example pipeline monitoring), connected vehicles, and traffic light systems [10].

There are some people that use the term fog and edge (see section 2.6 for more details) interchangeably, but these terms technically have two different meanings [73]. Edge computing devices are closer to IoT devices, while fog computing devices are a few network hops away from edge nodes. For example, an IoT gateway is a fog node, a J1939 ECU (see section 2.9.3) is arguably an edge node, and sensors and actuators are also edge nodes.

2.6 Edge Computing

Edge computing is performed on nodes that are in close physical proximity to data sources [73]. According to [73], there are three types of edge computing, namely: fog computing, mobile edge computing, and cloudlet computing. Typically edge computing has less resources than fog computing, since the edge of the network is located closer to the IoT devices.

2.7 IoT-enabling Technologies

2.7.1 Hardware

This section lists a series of hardware solutions that can be used in IoT deployment projects. Raspberry Pi, Intel Edison Board, and BeagleBone Black are mini computers that can be used as gateways, although the Intel Edison board has been discontinued (they no longer will support development). Arduino is a microcontroller that can be used as a gateway. Copperhill ECU simulator can be used as a bridge for reading and writing to J1939 networks. Bell Huawei E8372 Turbo stick is a Wi-Fi modem and Long-Term Evolution (LTE) modem, that provides a Wireless Local Area Network and an internet connection via LTE. Fona 800 provides a cellular network connection to a serially-connected device. UPS PIco provides the Raspberry Pi with uninterruptable power. Smart dust consists of small robots, that can be deployed non-invasively to many different areas. It supports wireless communication within a few millimeters and has the following applications: habitat monitoring, environment monitoring, traffic monitoring, security and asset racking, and smart health [72]. Anybus [4] is a hardware manufacturer for industrial IoT, and connects low level networks and devices to higher level network protocols (for example MQTT).

2.7.2 Physical and Data-link Layer Protocols

A literature review of [72] and [73] reveals the following: the Radio Frequency Identification (RFID) system is typically used to identify equipment without human intervention. This is accomplished by having an RFID tag on equipment, and a RFID reader to read the tag. RFID systems operate on 12V and 125 kHz. Wireless Identification and Sensing Platform (WISP) is similar to RFID, but does not require a battery. The reader is a high frequency RFID reader, which powers the WISP device wirelessly. It also supports a microcontroller that is programmable, unlike RFID. Wireless sensor networks (WSN) consist of small nodes with processing power, sensors, a battery, and wireless communication capabilities. Bluetooth Low Energy (LE) is designed to support transmission of small messages, and operates on the 2.4 GHz frequency. Near field communication uses magnetic field induction and has a small communication range (less than 10cm). Its frequency used is 13.56 MHz. IEEE 802.15.4 is a protocol on the physical and MAC layers, designed for low bit rate and low power wireless personal area network communication. IEEE 802.11AH enhances the IEEE 802.11 wireless standard (Wi-Fi). It operates on 900MHz, and is designed for low power consumption and less network overhead. Z-Wave is a low power MAC protocol designed for Home automation, operating on 908 MHz, a range of 30m, and has up to 100 Kbps bandwidth. The Long-Term Evolution Advance (LTE-A) (also known as 4g) has been developed in recent years by the 3rd Generation Partnership Project, which will enable IoT in the near future. It provide up to 100 Mbps.

2.7.3 Network Layer Protocols

According to [73], the Low Power Wide Area Network (LPWAN) allows resource-constrained devices to communicate over long range. A series of gateways are used to relay and spread data over the channels. IPv6 is used to overcome the lack of addresses of IPv4, and supports IPSec. IPv6 over Low Power Wireless Personal Area Networks (6LoWPAN) allows resource-constrained devices to access the internet via a wireless medium over IPv6. An edge router is needed to bridge the wireless network to the internet over IPv6.

Personal Area Networks are small networks, such as Bluetooth and Zigbee. Wireless Local Area Networks such as Wi-Fi provide higher bandwidth. Wide Area Networks cover large areas, such as cities. According to [72], the first cellular network coverage generation, 1g, or Analogue Cellular Telephony, appeared in the 1980s. These technologies continued to expand with new architectures. In 1998, Global System for Mobile Communication was invented from the advancement of cellular technologies. The Long-Term Evolution Advance (LTE-A), also known as 4g, has been developed in recent years by the 3rd Generation Partnership Project [73], which enables IoT. The 4g network can provide up to 100 Mbps using the LTE-A standard and 128 Mbps using the WirelessMan-Advanced, while 3g networks only provide up to 14.4 Mbps. 5g technology is being researched. The goal is to provide things and people with internet connection anywhere. 2g and 3g were not designed for the data demands that users have today.

According to [72], Vehicle ad hoc networks (VANETs) are technologies that allow vehicles to communicate with each other. Recent work, such as IEEE 802.11p and IEEE 1609, aims to allow vehicles to communicate with each other.

2.7.4 Machine to Machine Communication Protocol

According to [90], the gateways require a communication protocol that meets the requirements of low energy, low computational cost, and low latency, to enable gateways' software to connect to the cloud efficiently. Message Queue Telemetry Transport (MQTT) and Constrained Application Protocol (CoAP) are popular examples of machine to machine (M2M) protocols. They are based on the publish-subscribe architecture, which has the advantage that nodes that publish or subscribe to data do not need the identity of other nodes.

2.7.4.1 Message Queuing Telemetry Transport - MQTT

A literature review of [90], [47], [10], [73], [72], and [15] reveals that MQTT is an open-source M2M application layer communication protocol, which runs over the Transmission Control Protocol (TCP). It is computationally lightweight on the client-side and heavy-weight on the server side, which is ideal for resource-constraint IoT devices. It is designed to connect edge devices to applications and middleware. MQTT uses a publish-subscribe model, which is managed on the server-side by the MQTT broker. MQTT is based on message topics, such that when sending a MQTT message, a topic t is assigned to the

message. When a data producer publishes a message m, with the topic t, the message is sent to the MQTT broker. The MQTT broker will look up which client has subscribed to the topic t, and once the list of clients C is resolved, the MQTT broker broadcasts m and t to all clients found in C. Each subscriber will receive the message m and the topic t, unaware of the publisher's identify. MQTT has less packet overhead than the Hypertext Transfer Protocol (HTTP), and can vary the quality of service of each message. The MQTT broker does all the heavy computations. MQTT brokers can process thousands of published messages. MQTT is used by the Facebook mobile app for delivering messages and is used by Amazon Web Services for Amazon IoT cloud connection.

2.7.4.2 Constrained Application Protocol - CoAP

A literature review of [90], [73], [47], and [15] reveals that the Constrained Application Protocol (CoAP) is designed for devices with resource constraints. CoAP runs over the User Datagram Protocol (UDP) instead of TCP and uses a subset of HTTP commands. It was designed to overcome the limitations of HTTP to meet IoT requirements. CoAP is an important protocol that enables the growth of IoT research and development [23]. It is based on the Representational State Transfer (REST) architecture, and similarly to HTTP, uses responses and requests. Instead of using topics, it uses Universal Resource Identifiers (URI). That is, instead of publishing a message with a topic, publishers will publish to URIs. Subscribers will subscribe to these URIs by reading messages that are published to them.

2.8 Vehicle Maintenance Strategies

A literature review of [68], [103], [64], [106], [34], [35], and [75] reveals that there are two main categories of maintenance strategies, and some categories may include sub-categories. These categories can be found below:

1. Corrective/Reactive Maintenance
2. Preventive Maintenance
    (a) Time-based Maintenance
    (b) Condition-based Maintenance
        i. Proactive Maintenance
        ii. Predictive Maintenance

Corrective maintenance is performed after a component has failed. That is, in response to a fault that has occurred, the corrective maintenance strategy reacts and fixes it by replacing or repairing the broken components [103]. **Preventive maintenance**, starting in the 1970s [34], aims to prevent vehicle damage by quickly diagnosing vehicle faults

[106]. It can be separated into two categories, time-based maintenance and condition-based maintenance (which started in the late 2000s [34]). **Time-based maintenance** (TBM) replaces components preemptively, based on a schedule, to prevent unexpected component failures [75]. As a result, downtime is scheduled instead of being completely unexpected. The schedule is chosen based on the expected time before failure of the equipment [75]. **Condition-based maintenance** (CBM) uses diagnostic data to decide whether a repair is required in the vehicle's current state [106][34]. This way unnecessary repairs are avoided [75]. CBM enables real-time prognosis of equipment by monitoring the state of the equipment's health, which is achieved by analyzing sensor data that represents the physical state of the equipment [75]. Furthermore, a literature review of [34] and [35] reveals the following about CBM. Repairs are scheduled on vehicles only when the analysis of the selected sensors/parameters indicate maintenance is required. CBM can be divided into two subcategories, namely, proactive maintenance and predictive maintenance. **Proactive maintenance** monitors the equipment and attempts to identify/define threshold values of target sensors/parameters that best identify the condition of the equipment. Using these thresholds, the goal is to predict future vehicle failures, and the threshold values are adjusted when necessary. **Predictive maintenance** is similar to proactive maintenance, but it is more of an approach that is applied in a distributed computing environment. Predictive maintenance takes place after equipment diagnosis is performed. By reading sensor data and other diagnostic information, predictive maintenance attempts to analyze the condition (remaining useful life, degradation pattern, health status, etc.) of the equipment and to predict what type of maintenance is required to minimize costs and increase the equipment's lifetime. Predictive maintenance lowers costs by preventing failures, unscheduled maintenance, and downtime, and lowers costs by ensuring the replacement of failing parts is done only when needed [54][55][75].

2.9 Vehicle Diagnostics

2.9.1 Controller Area Network

A literature review of [94], [97], and [95] reveals that the Controller Area Network (CAN) protocol is a physical layer protocol, without any addressing. Instead it uses message identifications, which increase throughput by avoiding link message collisions. Lower message ids have higher priority. When electronic control units (ECUs) wish to send a message over a CAN bus, they begin sending the message immediately. To avoid packet collisions, bit arbitration is used. That is, ECUs analyze each bit they send over the wire by writing their bit on the wire and immediately reading from the wire. If the bit read from the wire is not the bit written to the wire, then ECUs contest the wire, that is, two different bits were written at the same time. The ECU with the higher bit backs off, and the ECU with lower bit continues to write the remaining bits of its message. Therefore, messages with the most zeros will win the wire contest. Message ids are used this way to give a message priority, regardless of the data found in the message's payload. The protocol is able to reach 1 Mbps. The protocol has no security mechanism by design.

2.9.2 On Board Diagnostics - OBD

According to [3], On Board Diagnostics (OBD) is a standard for accessing vehicle sensor data. The most recent version of the European standard is OBD-II, which is based on OBD [34]. This standard enables diagnostic software development by offering diagnostic trouble codes, sensor data, and the state of the vehicle [34]. The goal of OBD is to minimize the exhaust fumes produced by vehicles [34]. ELM-327, for example, is an adapter that reads OBD-II data and hosts a Bluetooth interface, which enables wireless sensor data reading. OBD is used for testing vehicle performance, by gathering data, storing it, and later analyzing it to perform maintenance decisions [34][64]. OBD is a standard that most modern vehicles use [64]. The sensor data values are transmitted over the CAN protocol and then evaluated by OBD-II [34].

2.9.3 J1939

The Society of Automotive Engineers' (SAE) J1939 protocol is based on the Controller Area Network (CAN) protocol. According to [94], [97], and [95], J1939 is a protocol for heavy-duty machinery. J1939 identifies sensors using a parameter group number (PGN) and a suspect parameter number (SPN). Sensors with lower PGNs have higher packet priority, which is due to bit arbitration performed by the CAN protocol (see section 2.9.1 for more information).

A J1939 network consists of electronic control units (ECUs), which manage sensors, actuators, and perform diagnostic tests for various mechanical components. ECUs are attributed unique ids based on the industry they are involved with. This field is called the *NAME* field, which is a 64-bit field that contains information about the ECU. Any ECU that writes to a J1939 network must be assigned a NAME field. An ECU that only reads from the network does not need a NAME field defined.

The sensor values are encoded in the 8-byte payload of the J1939 packet. J1939 implements a transport protocol in its protocol stack to support payload lengths of up to 1785 bytes. Since the CAN supports up to 8 bytes of data, large messages will be broken into multiple 8-byte CAN messages and will be reconstructed into the original message at the message's destination.

2.9.3.1 J1939 Specification Document

SAE has compiled a J1939 specification document, which lists all possible sensor types, including definitions, names, ids, and other information. Parameter groups, identified with a parameter group number (PGN), group sensors together. Sensors are defined as suspect parameters, identified with a suspect parameter number (SPN), and belong to a unique parameter group. Similar sensors are grouped together and share a PGN. Section 5.2.2 goes into detail on how to parse and use the J1939 specification document.

2.9.3.2 Parameters

When sensor readings (or parameters) are sent over the network, entire parameter groups are sent in one packet. That is, the value of each parameter of a parameter group is wrapped into a single packet and sent over the network, instead of sending a packet for each sensor reading. For example, the PGN 65262, which is defined in SAE's J1939 specification document, is the parameter group named *Engine Temperature 1*. It includes the following suspect parameters: *Engine Turbocharger 1 Oil Temperature*, *Engine Coolant Temperature*, and *Engine Fuel 1 Temperature 1*. When an ECU, E_1 needs the temperature of the fuel in the engine, E_1 will send a request to the ECU in charge of managing the engine's temperature, E_2. E_2 will then reply with a packet containing all the engine temperature information.

2.9.3.3 Parameter Encoding

The sensor values are encoded within the bytes of the J1939 messages. The J1939 specification document defines many fields/attributes for each parameter that can be used to decode and encode the sensors readings. Section 5.2.3 provides a complete example on decoding J1939 data.

2.10 Vehicle Prognostics

Prognostics is the prediction of the health status of monitored equipment. Prognostics can be achieved by analyzing failures modes, sensor data, product measurements, and diagnostic fault codes [13]. [53] states that prognostics and health management enable the following: a) health monitoring for faults and degradation of equipment, b) diagnosing odd behavior, c) analyzing remaining useful life, and d) event signaling when maintenance is required. There is an important distinction between prognostics and diagnostics. **Diagnostics** attempts to identify and isolate the occurrence of a fault once it has occurred, while **prognostics** attempts to predict a fault and degradation before they occur. Predicting the failure conditions for equipment is difficult. The motivation to perform vehicle prognostics is fleet management cost reductions. A single component that fails could affect many other components of a system [27]. One day of unexpected down time could cost a company many thousands of dollars. Having reliable industrial equipment is important for economical purposes of companies [75].

[27] mentions that prognostics can also be separated into offline and online approaches. The offline approach performs heavy-duty data analytics, which would require too many resources to be run on-board a vehicle. Information found using this approach is typically used for planning and management of the fleet, since it is usually not time critical information. The online approach is performed in real-time, which makes use of the real-time vehicle data to perform time-critical decisions. The information acquired from the online approach can be used to report imminent failures, to take preventive actions, and to re-schedule planning.

2.10.1 Methodologies

A literature review of [60], [75], [53], [106], [88], [27], and [64] reveals there are four types of approaches for prognostics: physical model-based, knowledge-based, data-driven, and the hybrid approach, which is any combination of the previous approaches.

2.10.1.1 Physical Model-based Methodology

A literature review of [75], [106], [64], and [87] reveals that the physical model-based methodology for equipment prognostics requires a deep understanding of the mechanical behavior of the equipment and of the physical models involved in the health of the equipment. Determining the remaining useful life of vehicle can be achieved using the physical model-based approach [27]. The models used in this methodology are composed of mathematical equations that represent the degradation of equipment and failure modes [27]. The parameters involved in these models are chosen from a large dataset. This methodology is appropriate when a mathematical model can be applied to the physical behavior of the equipment [53][62], which facilitates simulating faults. Statistical methods are used to detect when the equipment exceeds the expected behavior thresholds.

The advantage of this methodology, since it is tightly tuned to the equipment being monitored, are as follows: a) the models are generally simpler, b) the models are easier to understand, c) higher accuracy can be achieved [64][87], and d) it performs well in resource constrained environment (the fog for example), when applied to a simple system [27].

The disadvantages of this methodology are as follows: a) a deep understanding of the equipment and its behavior is required, which increases development costs [106]; b) if there are changes in the physical behavior of the equipment, the physical-model must also be updated [64]; c) the model-based approach performs poorly with complex systems [27]. Simplifications may be required to apply a physical-based model to a complex system, which would reduce the ability of the model to detect all faults [87]; and d) the ability of the model to predict failures is reduced when it is difficult to identify the physical nature of a fault or when there are too many variables to choose form [53].

2.10.1.2 Knowledge-based Methodology

According to [75] and [106], there are two main examples of the knowledge-based methodology, namely, expert systems (ES) and fuzzy logic (FL). ES represent domain expert knowledge in a programming model. It involves mapping the expert's reasoning into computer logic and rules [64]. FL is similar to Boolean logic, however it does not assign a variable the values 0 or 1, instead it assigns a range of confidence between 0 and 1.

The advantages of using FL are as follows: a) it enables uncertainty to be represented in the maintenance schedule, and b) it uses the membership of variables to make intuitive prognostic decisions. An advantage of using an ES is that knowledgeable decisions can be made using the defined rules.

The disadvantages of an ES are as follows: a) it is not trivial to add new rules, b) adding new rules also adds more computational overhead as the rule-base grows, and c) it requires a domain expert to define the rules.

2.10.1.3 Data-driven Methodology

According to [75], the data-drive prognostic methodology uses statistical and artificial intelligence (AI) approaches. These approaches use historical data and failure data [27][87][64] and can handle a variety of data that would be tedious to analyze using the other prognostic methodologies. Examples of the statistical approaches include the following: static and dynamic principal components analysis, linear and quadratic discriminant, partial least square, canonical variety analysis, Bayesian networks, hidden Markov modes, and regressive models. Examples of AI approaches include the following: polynomial neural networks, dynamic wavelet neural networks, self-organizing feature maps, and multilayer perceptron neural networks.

The advantages of this methodology are as follows: a) with the right historical data, error states or faults that would be too complex to simulate could be detected by simply observing historical data [87], b) less understanding of the physical system is required, since the decisions are purely data-driven, and c) data-driven models are more general than the models used by the other methodologies, since this methodology is purely data-driven, which means it can be applied in more contexts.

The disadvantages of this methodology are as follows: a) its dependency on the accuracy of the historical data [64], and the requirement of having access to failure data and data containing different modes of operation [27] [87], b) the internal behavior of this approach can be difficult to understand, since decisions are based purely on data [87], and c) this type of methodology requires a lot of computational resources, which means it would be difficult to deploy on the fog. It should instead be deployed on the cloud [27].

2.10.2 Data Characteristics

According to [106], there are two main categories for data, namely, event data and condition monitoring (CM) data. CM data represents the physical state of the equipment, while event data reports information about the equipment such as faults or failures. Other categories of data include, a) metadata, which is data about the equipment (for example, id, location, model, etc.), and b) business data, which describes the context the equipment is involved in, such as maintenance costs. It is important to consider that data may either be labeled or unlabeled. With labeled data, there is a history of data leading to failures. Unlabeled data indicates no failure data is available or observed in the dataset.

2.10.2.1 Condition Monitoring Data

CM data can be separated into different categories:hmm single values, time waveforms, and multidimensional data. Single values are single data points for a time unit. Time

waveforms are sets of time series data. Multidimensional data have multiple dimensions, for example images. The frequency at which the CM data is sampled can be categorized as follows: a) continuous, for data that is continuously being measured at some frequency, b) regular, where at a specific time interval it will be sampled, and c) irregular, which does not have a scheduled time when the data should be sampled. Data can come from multiple sensors or a single sensor. The CM data may be classified as heterogeneous or homogeneous.

2.10.2.2 Event Data

Event data can be categorized into the following categories: a) machine states, b) operating steps, such as the activity of equipment, c) configuration for the equipment settings, d) malfunctions, and e) maintenance actions such as the steps recorded to fix a malfunction. System behavior can also be categorized to help the data analytical process. Distinguishing continuous degradation from a sudden intermittent effect is important. Degradation may be monitored until a fault occurs or a sudden event halts (or deteriorates) normal equipment behavior. For sudden intermittent effect event data, it is important to determine if the event was only a temporary incident, or if the failure lead the equipment into a new machine state.

2.10.3 Challenges

A literature review of [104], [27], and [64] reveals the following challenges when performing vehicle prognostics: a) new unexpected faults typically occur between systems on board the vehicle; b) when new vehicles are created, they have new designs, which are prone to having bugs compared to older designs that have had more testing; c) some faults are infrequent and do not occur under the same conditions as other vehicles; d) the difficulty to predict the remaining useful life of equipment on board vehicles; e) electronic components typically fail instantaneously, instead of having gradual decay and degradation like mechanical equipment. Some electronic components also face degradation, which make it difficult to track the degradation when the component can instantly fail; f) oversensitive prognostics systems for vehicle produce false positive (FP) and false negative (FN) predictions, where FP predictions are false alarms and FN predictions are faults that failed to be predicted. This over sensitiveness is one of the main criticisms for vehicle prognostic systems; g) prognostic systems were engineered for the aerospace industry; vehicles in this industry have an enormous number of sensors, while commercial vehicles have nowhere near as many sensors, which make prognostics more difficult to implement on commercial vehicles; and h) the resource-constraints involved when performing on-board processing on vehicles.

2.11 Predictive Analytics

According to [50], predictive analytics involves using historical/training data to build models that are used on real data to make predictions and informed decisions. Applications of predictive analytics include the following: price prediction, dosage prediction, risk assessment, propensity modeling, diagnosis, and document classification.

According to [74], predictive analytics faces **challenges** when applied to big data, which include the following: a) variety of the data, b) concept drift, c) feature evolution, and d) concept evolution, that is, the data distribution changes over time [36].

A literature review of [106] and [74] reveals that multiple **strategies** can be used to perform predictive analytics, which include the following: a) summary statistics, which analyzes the behavior of equipment to learn interesting statistics. For example, counting faults and determining equipment efficiency; b) hypothesis testing, which consists of finding causal relationships; c) clustering, which is used to find novel concepts and allows heterogeneous groups of items to be grouped together by similarity. This can be used to group sensor data into fault categories, for example; d) classification, which can be used to categorize a variety of faults; e) anomaly detection, which can be used to find abnormal signals due to a fault or malfunction; f) frequent pattern mining, which finds correlations between items and could be used to find faults that occur together frequently; g) process mining, which reconstructs sequences of activities using event data and could be used for deviation detection; and h) sensor selection, which increases the accuracy of the chosen models.

2.11.1 Predictive Analytics Strategies in Fleet Management

A literature review of [57], [103], [84], [82], [79], [56], [83], [84], [87], [80], [46], [89], [91], [26], [104], and [83] is discussed in this section. Predictive analytics can be used to improve fleet management. There are two main categories of predictive analytics in fleet management, namely, fleet-wide and on-board predictive analytics.

On-board analytics are performed locally on a vehicle; that is, only a single vehicle's data is included in the machine learning models. This category has the following advantages: a) benefits from fog computing (see section 2.5 for more details); and b) supports simple models that only analyze individual vehicle's sensors, for example, expert systems or physical model-based approaches. **Fleet-wide analytics** involve analyzing and comparing the data of an entire fleet; that is, this category performs cross-fleet analysis. This category has the following advantages: a) it can more accurately detect faults, b) it can more accurately identify no-fault-founds, c) it can identify new types of faults, which were previously unknown, by comparing the similarities between vehicles' diagnostics data, d) it can identify the root cause of the faults; and e) it works well for monitoring the normal behavior of many similar vehicles (a popular scenario), since fault data is usually more difficult to obtain and a fleet of similar vehicles is common in the industry. Furthermore, by combining fleet-wide and on-board analytics, more accurate predictive maintenance systems can be deployed.

Common **strategies** used to perform predictive analytics for fleet management include the following: a) degradation analysis; b) physical model analysis; c) sensor selection, which can be done using expert knowledge or using data mining strategies. Note that vibration sensors are a popular choice for predicting failures; d) separating the vehicles into subsystems, which can be done using divide-and-conquer techniques. Simple machine learning models can then be applied to each subsystem, instead of deploying a single complex machine learning model for the entire system; e) pattern recognition, which involves detecting faults either by analyzing a recorded history of faults or by analyzing a history of healthy behavior; that is, when the current vehicle behavior matches (or deviates from) the signature of either history, then faults can be predicted; f) outlier/anomaly detection, which involves comparing sensor values together and attempting to detect when a sensor deviates from the others. This strategy reveals odd behavior, faults, and faulty sensor values and is typically performed in the cloud; g) fleet-wide analysis, which involves analyzing a fleet of vehicles to helps detect faults that would have been otherwise undiscovered if only sensor data from individual vehicles was analyzed; and h) using diagnostic trouble codes, which can be used to label fault data without needing a mechanic to manually label a dataset. Note that diagnostic trouble codes are used to activate diagnostic lamps on-board the vehicles, for example, the "check engine light".

When either the equipment's health status is estimated, events are detected, or other interesting discoveries are made, these findings should be presented visually in a user-friendly way to a mechanic.

2.11.2 Sensor Selection

Sensor selection is important for determining the health of equipment [54][60]. In this section the terms sensor, signal, parameter, and feature will be used interchangeably, and all refer to a stream of data that can be analyzed by a machine learning model (MLM). Having many sensors on equipment is helpful for conducting prognostics and condition monitoring, since without data, predictive analytics would not be possible. The sensor selection choice has an effect on the accuracy of the predictions made by the MLM [60]. In an IoT environment, since processing power is one of the resource constraints, care must be taken when selecting sensors. Increasing the number of selected sensors will proportionally increase the amount of processing power required by the MLM [60]; that is, one cannot naively include all the sensors in the MLM. Therefore, it is essential to carefully choose the best features to include.

A literature review of [13], [54], [55], [82], [60], [69], [68], [102], and [98] reveals the following sensor selection strategies: a) correlating sensor data with the accumulation of degradation and damage to a system, which may involve performing fault injection, analyzing labeled fault data, or gathering labeled fault data; b) installing sensors on equipment that lack a sufficient number of sensors; c) using the principal component analysis method, which reduces the dimensions (number of sensors) of a dataset, and attempts to minimize information loss and retain the important relationships between sensors; d) using an expert-system to select sensors; e) using an entropy-based sensor selection approach; f) using expert knowledge of critical components to select sensors; g) using fleet-wide analytics

to select sensors; h) using on-board analytics for sensor selection; i) using the *Unsupervised Embedded Algorithm* proposed by [69], which is based on entropy, uses the K-Means algorithm (an unsupervised clustering algorithm), and ranks a list of filtered sensors instead of including all the available sensors in the MLM; j) using mutual information between sensors while using entropy; and k) using Failure Mode Effects Analysis;

2.12 Statistics

2.12.1 Probability Distributions

According to [50], an important process in predictive analytics is getting to know the data one is working with. By analyzing the mean, standard deviation, minimum, maximum, histogram, and other summary statistics of the sensor data, this will provide on with insight into the probability distribution of a target dataset. A histogram can be used to approximate the probability distribution of a dataset, and it can be used to predict the probability that a data instance will fall into a target histogram bin [67]. Below is a list of common probability distributions:

- **Uniform Distribution** consists of data instances that are equally likely to have any value within a specific range. For example, a dataset of unique identification numbers could follow such a distribution. Figure A.1 illustrates the probability density function of a uniform distribution.

- **Normal Distribution** consists of data instances that tend to take on a central value and have a symmetric variation away from the center. For example, the heights of random men in a population could follow such a distribution. Normal distributions can also be referred to as unimodal (a single central peak).

  An important normal distribution that has many applications in data science is the standard normal distribution, which is a normal distribution with mean = 0 and variance = 1 [67]. Figure A.2 illustrates the probability density function of a normal distribution.

- **Skewed/Beta Distribution** consists of data instances that tend toward a left or right peak. Figure A.3 illustrates the probability density function of a beta distribution.

- **Exponential Distribution** consists of a distribution where the majority of its data instances tend toward very high or very low values. The likely hood of finding data instances far away from the left or right central tendency decreases at an exponential rate. Such distributions often have outliers. Figure A.4 illustrates the probability density function of an exponential distribution.

- **Multimodal distribution** consists of a distribution where the data instances tend towards multiple central tendency peaks. A bi-modal (or binomial distribution) is

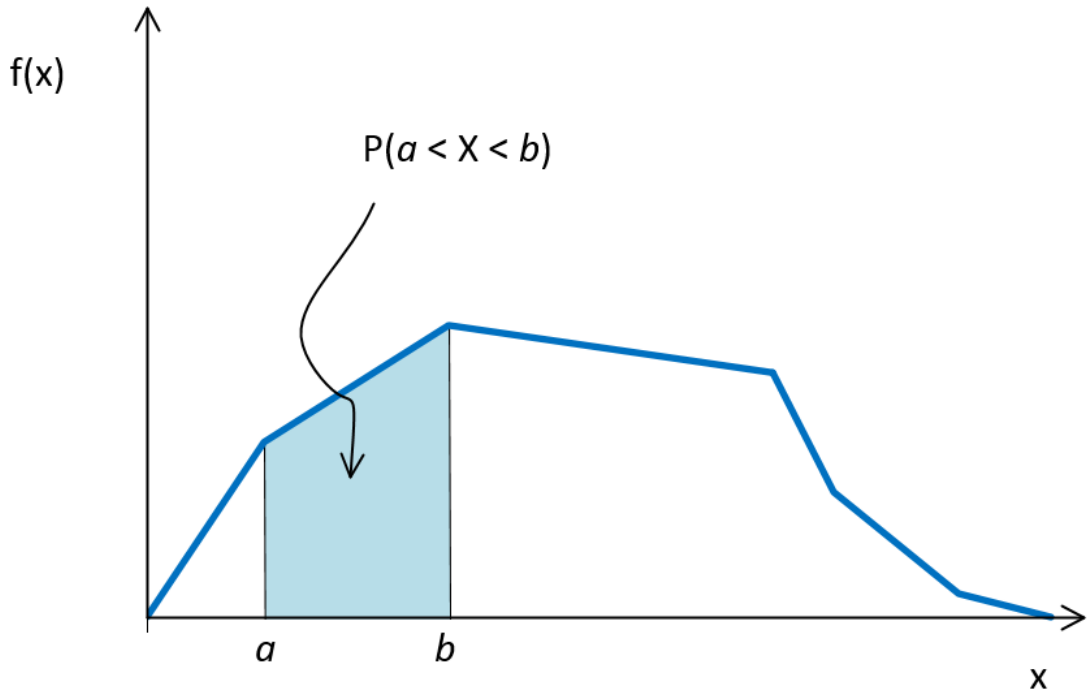

Figure 2.1: Probability of a random variable X of falling in the area between a and b under the curve f(x), inspired from [67]

an example of a multimodal distribution and can be conceptualized as two combined normal distributions with different peaks. For example, sampling the height from a population of men and woman could follow a bi-modal distribution. Figure A.5 illustrates the probability density function of a bi-modal distribution.

2.12.2 Probability Density Functions

According to [67], probability density functions describe the probability distribution of a continuous random variable X. They are used to determine the probability that the value of X is found within some interval by calculating an area under their curve at a given interval range. Figure 2.1 illustrates an example of determining the probability that a data sample X falls between the intervals a and b; that is, $P(a < X < b)$. Probability density functions have the properties found in equation 2.1.

$$\begin{aligned} &(1) && f(x) \geq 0 \\ &(2) && \int_{-\infty}^{\infty} f(x)dx = 1 \\ &(3) && P(a \leq X \leq b) = \int_{a}^{b} f(x)dx = \text{area under } f(x) \text{ from } a \text{ to } b \end{aligned} \tag{2.1}$$

2.12.3 Sampling Data

At times it is necessary to analyze only a subset of a dataset; that is, a sample of the dataset. According to [67], when sampling data from a distribution with a known standard deviation, it may be desirable to know how close the sample mean $\overline{x}$ (the mean of the set of random variables sampled) is from the actual mean. A few definitions are required to explain how this can be achieved.

A normal random variable with $\mu = 0$ and $\sigma^2 = 1$ is called a standard normal random variable, and is denoted as Z, where μ is the mean and σ^2 is the variance. Determining the probabilities of these random variables is a typical task in statistics, which is why many software packages offer this functionality. Probabilities are typically in the form of P(Z $\leq z$), for some random variable z. Probabilities of the form P($z_1 < Z < z_2$) = P(Z $< z_2$) - P(Z $< z_1$).

When n random variables X_1, X_2, ..., and X_n are sampled from a distribution, it is called a **random sample**. Since all the X_i's are random variables, then the sample mean $\overline{X}$ is also a random variable. According to the central limit theorem, which is expressed in equation 2.2, if we were to repeatedly sample n samples from a large dataset with mean μ and variance σ^2/n, then the distribution of the sample mean would be normal. As n tends towards ∞, the sample mean distribution is the standard normal distribution.

$$\begin{aligned} Z &= \frac{\overline{X}-\mu}{\sigma/\sqrt{n}} \\ n &\to \infty \end{aligned} \tag{2.2}$$

In other words, for any dataset with an arbitrary distribution, if the dataset has enough data instances, we can estimate the probability that a sample mean be within a certain interval.

Example 1.

A thermometer sensor's mean temperature reading is 100 °C, with standard deviation of 10 °C, and the thermometer sensor has an arbitrary distribution. To find the probability that $n = 25$ sensor readings will have an average temperature of less than 95 °C, the following calculation is done. Since the sampling distribution $\overline{X}$ is normal, with mean $\mu_{\overline{X}}$ = 100 °C, and standard deviation of:

$$\sigma_{\overline{X}} = \frac{\sigma}{\sqrt{n}} = \frac{10}{\sqrt{25}} = 2,$$

Standardizing the point $\overline{X} = 95$, then we have:

$$z = \frac{95-100}{2} = -2.5,$$

and therefore:

$$P(\overline{X} < 95) = P(Z < -2.5) = 0.0062,$$

In other words, the probability that the sample mean is less than 95 is 0.0062.

Example 2.

This example expands on the Example 1 above. To determine the probability that the sample mean falls within 5% of the actual mean, that is, $95 < \overline{X} < 105$, apply the central limit theorem expressed in equation 2.2 twice to calculate $P(\overline{X} < 95)$ and $P(\overline{X} < 105)$. Remember that $P(95 < \overline{X} < 105) = P(\overline{X} < 105) - P(\overline{X} < 95)$. Re-apply the same steps as Example 1 above to find $P(\overline{X} < 95) = 0.0062$ and do the same to determine $P(\overline{X} < 105)$:

$$\begin{aligned} P(95 < \overline{X} < 105) &= P(\tfrac{95-100}{10/\sqrt{25}} < Z < \tfrac{105-100}{10/\sqrt{25}}) \\ &= P(\tfrac{95-100}{2} < Z < \tfrac{105-100}{2}) \\ &= P(-2.5 < Z < 2.5) \\ &= P(Z < 2.5) - P(Z < -2.5) \\ &= 0.993790 - 0.0062 \\ &= 0.98759 \end{aligned}$$

Therefore, the probability that the sample mean is within 5% of the actual mean is 0.98759, for a sample size of 25.

2.12.4 Histograms

Histograms can be used to approximate the probability that a data instance falls within a certain interval by using the area of the bin that the data instance falls under. In other words, the bins of the histogram are the proportional frequency of data instances found in a dataset [67]. [43] mentions a couple of statistical methods for determining bin size of histograms, namely Scott's rule and the Freedman and Diaconis rule. **Scott's rule** is expressed in equation 2.3:

$$h = 3.5s\sqrt[3]{n} \tag{2.3}$$

, where s is the standard deviation of the dataset used to create a histogram, n is the number of samples in the dataset, and h is the bin width.

The **Freedman and Diaconis rule** is expressed in equation 2.4:

$$\mathrm{h} = 2\frac{\mathrm{IQR}(x)}{\sqrt[3]{n}} \tag{2.4}$$

, where IQR is the interquartile range (see section 2.12.5) of the dataset of size n, and h is the bin width. The statistical details of Freedman and Diaconis rule are discussed in detail in [33]. The Freedman-Diaconis rule attempts to minimize the difference between the area under the curve of the empirical probability distribution and theoretical distribution [99].

To determine the number of bins, it can be computed using equation 2.5 once the bin size h has been computed. Where max is the maximum observed value in a dataset and

min is the smallest observed value in a dataset. In an IoT environment this is convenient, since when working with sensor values, the sensor maximum and minimum output values are typically defined in a sensor definition database [82].

$$n = (max - min)/h \tag{2.5}$$

2.12.5 Interquartile Range

According to [67], the interquartile range (IQR) is a measure of variability in a dataset, which is defined by measuring the difference between quartile one and three. The IQR is defined as follows [67]:

$$IQR = q_3 - q_1$$

When a dataset is sorted and partitioned into four equal parts, three quartiles are created. The first quartile, q_1, has approximately 25% of the samples below it, and therefore, about 75% of samples are above it. The second quartile, q_2, has approximately 50% of the samples above and below it. The third quartile, q_3, has approximately 75% of values below it, and 25% above it. Note that the IQR is resistant to outliers.

2.12.6 Entropy

According to [82] and [59] entropy defines the amount of uncertainty (or randomness) that exists in a dataset. Entropy can be defined as follows in equation 2.6:

$$E = -\sum_{i=1}^{N} p_i(x) log(p_i(x)) \tag{2.6}$$

2.13 Anomaly Detection

Anomaly detection, or deviation detection, attempts to detect when an instance deviates from its norm [106][81]. According to [105], there are four basic methods for detecting outliers: counting objects in k-neighborhoods, kth nearest neighbor distances, outlier scores, and comparing an object's neighbor to its neighbor.

At times it may be necessary to rank how much a point is deviating from the norm by using outlier scores; that is, by estimating a p-value for a data instance. [81] mentions that estimating outlier scores is used less frequently than deciding whether a sample is an outlier or is normal. Outlier detection is sensitive to dataset size, which can be tackled by dimension reduction techniques, which is similar to sensor selection (see section 2.11.2). Evaluating outlier detection algorithms can be accomplished measuring the area under a receiver operating characteristic (ROC) curve.

2.13.1 Evaluation of Anomaly Detection Algorithms Using ROC Curves

A literature review of [59], [30], [14], [22], [63], [37], and [32] is explained in this section. ROC curves are used to compare the performance of anomaly detection algorithms, by comparing the area under the curve (AUC) of a ROC curve. A ROC curve point is a true positive rate (TPR) and false positive rate (FPR) pair, which are associated to a threshold value used to create the pair. As such, ROC curves can be analyzed to help decide what threshold value should be used when a specific FPR is required for a target system.

2.13.1.1 Evaluation Metrics

A few definitions are required to help explain the important notions of this section. Let data instances with a negative label and those with a positive label be defined as normal and as an outlier (or an anomaly), respectively. Anomaly detection algorithms attempt to predict the labels correctly. When evaluating an anomaly detection algorithm, labeled training data is required to evaluate the predicted labels. For example, suppose a vehicle v_1 suffers from a fault and a vehicle v_2 has no faults. The actual label of v_1 is positive (an anomaly), and the actual label of v_2 is negative (normal). A positive prediction for v_1 is true positive (TP), a negative prediction for v_1 is false negative (FN), a positive prediction for v_2 is false positive (FP), and a negative prediction for v_2 is true negative (TN).

There are six metrics that can be used to compare anomaly detection algorithms (which use a subset of TP, FP, FN, and TN in their calculations), namely: false positive rate (FPR), false negative rate (FNR), true positive rate (TPR), true negative rate (TNR), accuracy (ACC), and F-score. FPR and TPR are the most commonly used metrics when creating a ROC curve, since when combined, they perform well when there are considerably more data samples of one class than another class, which is often the case. Smaller FPR and FNR values indicate better anomaly detection performance. Some of these metrics can be found below:

- **False Positive Rate** refers to the ratio of data instances falsely flagged as anomalies compared to the total number of normal data instances, and is defined as follows in equation 2.7:

$$FPR = \frac{FP}{FP+TN} \tag{2.7}$$

  , where FP is the number of non-outlier data instances flagged as outliers, and $TN + FP$ is the total number of normal data instances.

- **False Negative Rate** refers the ratio of undetected outlier data (anomaly data classified as normal) compared to the total number of outlier data instances, and is defined as follows in equation 2.8:

$$FNR = \frac{FN}{FN+TP} \tag{2.8}$$

, where FN is the number of anomalies flagged as normal, and $FN + TP$ is the total number of anomalies.

- **Accuracy** refers to the ratio of properly labeled/identified data instances (either normal or outlier) compared to the total number of data instances detected, and is defined as follows in equation 2.9:

$$ACC = \frac{TP+TN}{FP+FN+TN+TP} \tag{2.9}$$

, where $TP + TN$ is the total number of data instances properly labeled, and $FP + FN + TN + TP$ is the total number of data instances.

- **True Positive Rate** refers to the ratio of outlier data instances correctly classified as outliers compared to the total number of outlier data instances, and is defined as follows in equation 2.10:

$$TPR = \frac{TP}{TP+FN} \tag{2.10}$$

, where TP is the number of outlier data instances correctly flagged as outliers, and $TP + FN$ is the total number of outlier data instances. Note that in some literature they refer to the TPR as the detection rate, sensitivity, or recall.

- **F-score** provides an overview of the average performance of the outlier detection algorithm, and is defined as follows in equation 2.11:

$$F - score = \frac{TP}{TP+(FP+FN)/2} \tag{2.11}$$

2.13.1.2 Computing a ROC Curve

To compute a ROC curve the following is required: a) an anomaly detection algorithm (ADA) that outputs results in the form of outlier scores, and b) a labeled training dataset, T. Let T_i denote the ith training data sample, where $T_i = 0$ for instances with a negative label and $T_i = 1$ for instances with a positive label. Using dataset T, the ADA is run and produces the outlier score result set O, where $\forall z \in O$, $z \in \mathbb{R}$, $0 \leq z \leq 1$, and $|O| = |T|$. Notice that the outlier scores in O are not labeled, and to compute a ROC curve point p the TPR and FPR are required. To compute the TPR and FPR, $\forall z \in O$ must be assigned an estimated label. Let $L(z, t)$ denote the estimated label for the outlier score $z \in O$, for some threshold $t \in \mathbb{R}$, where $0 \leq t \leq 1$. The TP, FN, FP, and TN rates are computed as follows in equation 2.12:

Sample	1	2	3	4	5	6	7	8
Score	0.75	0.05	0.15	0.2	1	0.4	0.3	0.45
Actual Label	F	F	T	T	T	T	F	T
Sample	9	10	11	12	13	14	15	16
Score	0.5	0.85	0.55	0	0.55	0.9	0.95	0.25
Actual Label	T	T	F	T	T	F	F	F

Table 2.1: Computing a ROC curve example: labeled samples and outlier score for 16 samples, where "F" = False and "T" = True.

$$
\begin{aligned}
L(z,t) &= \begin{cases} 0 \text{ (negative prediction)}, & \text{where } z > t \\ 1 \text{ (positive prediction)}, & \text{where } z \leq t \end{cases} \\
\text{TP} &= \sum_{i=1}^{|O|} O_i \cdot L(O_i,t) \\
\text{FP} &= \sum_{i=1}^{|O|} \big(1 - O_i\big) \cdot L(O_i,t) \\
\text{FN} &= \sum_{i=1}^{|O|} O_i \cdot \big(1 - L(O_i,t)\big) \\
\text{TN} &= \sum_{i=1}^{|O|} \big(1 - O_i\big) \cdot \big(1 - L(O_i,t)\big)
\end{aligned}
\tag{2.12}
$$

Compute the TPR and FPR using equation 2.10 and 2.7, respectively, to create a ROC curve point $p = \{TPR, FPR\}$, for some threshold t. To create all the ROC points, apply the steps mentioned above for all thresholds. To define the set of thresholds $\hat{T}$, the following steps are performed:

1. Sort O from smallest to largest. Let $\hat{O}$ denote the sorted set.
2. Remove duplicate values from $\hat{O}$.
3. Let $\hat{T} = \hat{O}$

Re-apply the steps mentioned above $\forall t \in \hat{T}$ to compute all points of a ROC curve. For completeness, if not already present in the resulting set of points, the points (FPR = 0, TPR = 0) and (FPR = 1, TPR = 1) must be added.

Computing a ROC Curve Example

A complete example of computing a set of ROC curve points is found in this section. Let table 2.1 define the label assignment of the samples from the training dataset, and the outlier scores for each sample computed by the ADA; where the *Sample* row indicates the sample id, the *Score* row indicates the outlier score output by the ADA, and the Actual Label row indicates the label assigned to the sample in the training dataset. To compute the FPR and TPR, and to compute the sets O, $\hat{O}$, and $\hat{T}$, apply the steps mentioned above in section 2.13.1.2. After performing these steps, the elements of the resulting sets are illustrated below:

Sample	1	2	3	4	5	6	7	8
Score	0.75	0.05	0.15	0.2	1	0.4	0.3	0.45
Actual Label	F	F	T	T	T	T	F	T
Estimated Label	F	T	T	T	F	T	T	T
Result	TN	FP	TP	TP	FN	TP	FP	TP
Sample	9	10	11	12	13	14	15	16
Score	0.5	0.85	0.55	0	0.55	0.9	0.95	0.25
Actual Label	T	T	F	T	T	F	F	F
Estimated Label	F	F	F	T	F	F	F	T
Result	FN	FN	TN	TP	FN	TN	TN	FP

Table 2.2: Computing a ROC curve example: evaluation results for threshold = 0.45, where "F" = False, "T" = True, "FN" = False Negative, "TN" = True Negative, "TP" = True Positive, and "FP" = False Positive.

- O = {0.75, 0.05, 0.15, 0.2, 1, 0.4, 0.3, 0.45, 0.5, 0.85, 0.55, 0, 0.55, 0.9, 0.95, 0.25}
- $\hat{O}$ = {0, 0.05, 0.15, 0.2, 0.25, 0.3, 0.4, 0.45, 0.5, 0.55, 0.55, 0.75, 0.85, 0.9, 0.95 , 1}
- $\hat{T}$ = {0, 0.05, 0.15, 0.2, 0.25, 0.3, 0.4, 0.45, 0.5, 0.55, 0.75, 0.85, 0.9, 0.95, 1}

To compute the ROC curve points, $\forall t \in \hat{T}$, count the TP, FP, FN, and TN rates by applying equation 2.12 and compute the FPR and TPR. For example, when using threshold = 0.45, Table 2.2 illustrates the TP, FP, FN, and TN predictions. In this case, TP = 5, FP = 3, FN = 4, and TN = 4. The FPR and the TPR are computed as follows:

$$\text{TPR} = \frac{TP}{TP+FN} = \frac{5}{5+4} = \frac{5}{9} = 0.555$$
$$\text{FPR} = \frac{FP}{FP+TN} = \frac{3}{3+4} = \frac{3}{7} = 0.429$$

2.13.1.3 Interpreting a ROC Curve

A ROC curve's x-axis is the FPR and the y-axis is the TPR, which can be seen in figure 2.2. To obtain a point (p,p), the classifier (that is, the point created by applying a threshold to an ADA's outlier score result set) must predict positively with probability p, and predict negatively with probability (1-p). The point (0,0) represents the classifier that always makes negative predictions, and (1,1) represents the ADA that always makes positive predictions. The point (0,1) is the best classifier, which predicts everything correctly, and the point (1,0) is the worst classifier, which predicts everything incorrectly. The diagonal line (l_1), from the point (0,0) to point (1,1), represents a random classifier. Classifiers found above l_1 predict more accurately than random, while those below l_1 predict more poorly than random. The diagonal line (l_2), from (0,1) to (1,0), are classifiers that perform equally well when correctly predicting negative and positive classes for data instances. Classifiers that are found below l_2 predict negative classes with better accuracy, and classifiers found above l_2 predict positive classes with better accuracy.

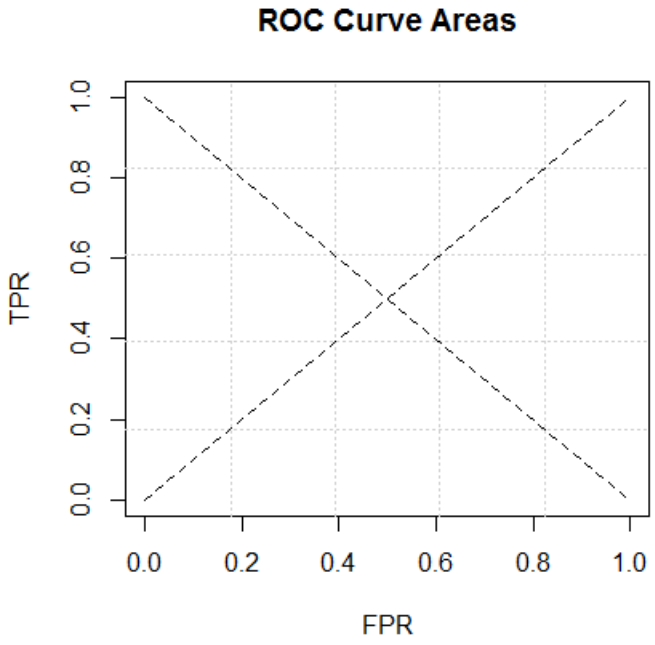

Figure 2.2: ROC curve true positive rate vs. false positive rate, inspired by [32]

2.13.1.4 ROC Curve Point Interpolation

A few definitions are required for notation simplicity. Let O denote the set of outlier scores produced by an anomaly detection algorithm. Let p_i denote the ith ROC curve point (x_i, y_i), where x_i is the FPR of p_i and y_i is the TPR of p_i. Let t_i denote the ith threshold used on the outlier scores of O to produce the point p_i.

To perform interpolation, although it may be tempting to draw a line, $l = \{p_i, p_j\}$, between p_i and p_j, for some $j > i$, it is not necessarily the case that all points from p_i to p_j will fall on the line l. In other words, for some $k = j - i$ and $z = i + \frac{k}{2}$, it is not necessarily the case that t_z produces a point p_z that will be found on l. However, there is a technique that can ensure this interpolation is correct.

Let O^1 and O^2 denote subsets of outlier scores randomly sampled from O, where O^1 and O^2 have the following properties:

$$
\begin{gathered}
|O^1| \subseteq O \\
|O^2| \subseteq O \\
|O^1| = \sigma \cdot |O| \\
|O^2| = (1 - \sigma) \cdot |O| \\
O^1 \cup O^2 \equiv O \\
O^1 \cap O^2 \equiv \emptyset \\
\sigma \in \mathbb{R},\ 0 \leq \sigma \leq 1
\end{gathered}
$$

, where σ is used to represent the fraction of elements from O that each subset O^1 and O^2 contain. In this case, to create a point (p), half-way between p_i and p_j, let $\sigma = 0.5$ when

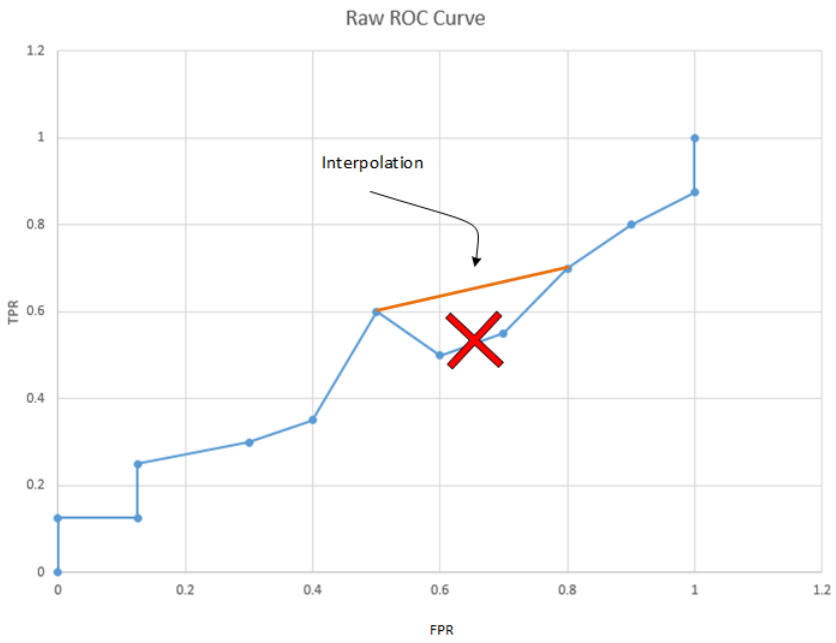

Figure 2.3: ROC curve point interpolation example, where the red X indicates which points are removed, and the orange line represents the interpolation from the point (0.5,0.6) to the point (0.8,0.7).

generating the subsets O^1 and O^2. To guarantee that p is found on the line l, apply t_i to the outlier scores in O^1 and apply t_j to the outlier scores in O^2 when creating p. This technique can be used to generate all points on the line l, by applying the steps mentioned above $\forall \sigma \in [0, 1]$. Note that in practice, this can be achieved by applying the thresholds t_i and t_j with probability σ and $(1 - \sigma)$, respectively, whenever an anomaly detection algorithm makes a prediction.

In fact, although it is mathematically correct to include all points in a ROC curve, in practice most data analysts remove points from the ROC curve that decrease the TPR by using the point interpolation technique mentioned above. An example of this is found in figure 2.3

2.14 Document Retrieval

Document retrieval is another form of machine learning. According to [39] and [44], document retrieval attempts to fetch objects that are relevant to a query. [39] discusses in detail the statistical interpretation of recall, precision, and f-score. They define precision as *"the probability that an object is relevant given that it is returned by the system"*, and define recall as *"the probability that a relevant object is returned"*. An example of document retrieval is an online search engine that fetches links when a user searches for content. Table 2.3 illustrates the label assignment of document retrieval algorithms, and equation 2.13 defines the precision and recall in terms of relevant and irrelevant objects. Objects that

Actual Label ω / Prediction ν	ν: relevant	ν: irrelevant
ω: relevant	TP	FP
ω: irrelevant	FN	TN

Table 2.3: Document retrieval: object label assignment based on relevant or irrelevant objects retrieved.

are fetched by document retrieval algorithms are predicted as relevant, and the objects that were not fetched are predicted as irrelevant to the search query.

$$\begin{aligned} p &= \frac{TP}{TP+FP} = \frac{\texttt{number of relevant objects predicted}}{\texttt{total number of actual relevant objects}} \\ r &= \frac{TP}{TP+FN} = \frac{\texttt{number of relevant objects predicted}}{\texttt{total number of objects predicted as relevant}} \end{aligned} \tag{2.13}$$

2.15 Comparisons - Main Approaches and Classification

2.15.1 Approach 1 - Estimating p-Values for Deviation Detection

[81] proposes a data-driven predictive maintenance approach. They attempt to estimate deviation scores for each vehicle in a fleet of city buses. A more complete version of this work is found in [71], which contains all the details of the experiments done in [81].

A deviation score assigned to an object represents the amount the object is deviating from the rest of its distribution. Deviation scores are useful, since they typically range from 0 to 1 and therefore can be interpreted as p-values. They ([81]) attempt to estimate the scores of each vehicle by comparing each vehicle to the rest of the fleet. Their null-hypothesis is that each type of sensor outputs the same distribution on each vehicle. In other words, two sensors of the same type should output the same distribution even if both are found on two different vehicles. Therefore, computing the distance between two models (used to sample from sensors of the same type) should produce a p-value that is uniformly distributed between 0 and 1.

They select three methods for producing these p-values, namely: one class support vector machine (OCSVM), conformal anomaly detection (CAD), and most central pattern method (MCP). To produce p-values, for each of these algorithms, they compute a distance matrix that represents the distance between each model, where each row represents a model and each column also represent a model; these models can be from any of the vehicles in the fleet. The cells in the matrix contain the distance between two models, with respect to the cell's column and row, which can be calculated using an appropriate distance measure depending on the type of model.

2.15.1.1 Distance Measures

In their experiments, they use two different kernels for the OCSVM: the standard Gaussian kernel for their simulated data, and a Hellinger kernel (which corresponds to the Hellinger distance between two histograms) for their live vehicle data. They use the k nearest neighbor (kNN) non-conformity measure for the CAD experiments, and for the MCP method, they find the most central model by using the Euclidean distance. A p-value for a test model is estimated by computing a ratio of data samples farther away from the central model than the test model.

2.15.1.2 Experimental Data

They experiment on simulated data and on live data, which they gathered from a fleet of city buses. In their **simulated data** experiments, they generate data with a variety of distributions, namely: Normal, Beta, and Student's T, Lognormal, Exponential, Elliptic Multivariate Normal, and Bi-modal. They run experiments 30 times for each distribution to ensure average behavior is captured. The **live data** they gathered was from a city bus fleet in Sweden, which was collected over years. They use GPS sensor readings to determine when the bus was en route and when it was in the shop. They sample data from J1587 networks using read-only data samplers, which they installed on each bus. Their gateways would upload histograms of the data to a database via a wireless internet connection. They use the Hellinger distance to compute distance between histograms.

2.15.1.3 Results

In their results, they found which p-value estimation method works best for each distribution. They provide a use case for estimating p-values to build a system of a self-monitoring fleet that increased the average vehicle up-time.

2.15.1.4 Advantages

The advantages are as follows: a) their approach supports fleet-wide analytics; b) they only send models (aggregates of the sensor readings), from the gateway to the back-end server instead of sending the raw signals. This means they save a lot of bandwidth, since models generally abstract the raw signal and reduces the size of the data; c) their experiments are done on simulated as well as live data; and d) they compliment their work with the use case of increasing average vehicle up-time for a fleet.

2.15.1.5 Drawbacks

The drawbacks are as follows: a) they do not have access to the raw sensor readings when performing experiments. They only had access to histograms, which means they may have lost important information during their studies. Although it is a good approach to only

send aggregates of the sensor readings to a server from each vehicle, for experimental purposes having access to the raw datasets is important, so the models, in this case histograms, do not capture all behavior of the signals. For example, histograms do not capture the time series aspect, behavior over time, of the sensor signals. b) They sample sensor data only at a rate of 1Hz, which means they may have lost valuable information that could only be captured by sampling at a higher frequency.

2.15.2 Approach 2 - A Field Test With Self-organized Modeling for Knowledge Discovery in a Fleet of City Buses

[18] propose a data-driven predictive maintenance approach. They monitor the signals over the controller area networks of a fleet of city buses. They attempt to find the most interesting signals, and use maintenance records to make informed decisions. In other words, they propose a semi-supervised machine learning approach by combining the data-driven approach (unsupervised signal modeling), combined with external information sources (a vehicle service record database). They perform on-board analytics on each bus. They mention that histograms are computationally cheap, which suggests they are candidates for on-board analytics, due to the resource constraints. They claim that the Bhattacharyya distance and "symmetry index" are good metrics for comparing histograms, since they are computationally cheap, and mention the Earth Mover Distance, which requires significantly more computations. They define an interesting signal as a non-random signal.

A signal's randomness and histogram distance are metrics used to measure the interestingness of signals on each bus. They perform fleet-wide analytics, and define a vehicle that deviates from the rest of the fleet as abnormal, since they assume by default the fleet is healthy (normal). They point out three strategies for applying external knowledge to the unsupervised fleet-wide analytics: vehicle database relation mining, cloud-based diagnosis software, and simulation-based mining.

2.15.2.1 Experimental Data

They had a fleet of 18 city buses and read signals from the CAN network. The models they use to aggregate the sensor data were histograms and linear models. They had access to vehicle services records of the entire fleet, which they use to estimate repair dates. They collect hourly histograms for each signal on all the vehicles. They use the entropy of the histograms to measure interestingness of signals.

2.15.2.2 Deviation Detection

They assign a health status to vehicles weekly, by comparing the distances of groups of histograms to create an empirical distribution of distance measures. When 5%, 50%, and 75% of a vehicle's histograms are outliers, the vehicle is flagged with a *warning*, *orange alert*, and *red alert* flag, respectively. They use the Bhattacharyya distance for histogram

distance computations. The second model they use are linear models. They gather data for a 30 hour period of 12 signals, and analyze all the pairwise combinations of these signals, seeking the combinations of signals that have the strongest relations (see equation 2.14).

$$x_1 = b_1 x_2 + b_2 \tag{2.14}$$

Where x_1 and x_2 are the signal pairs, b_1 is the slope, and b_2 is the intercept. A signal pair with a small mean square error fleet average has a strong relation. What is interesting is when a vehicle's b_i values deviate from the rest of the fleet it suggests a fault has occurred. They use the dates of the deviations to try to cross-reference a vehicle service record database to diagnose the possible fault.

2.15.2.3 Advantages

The advantages are as follows: a) their approach supports resource management, since they perform on-board analytics; b) their approach supports fleet-wide analytics; c) they deploy a non-intrusive predictive maintenance prototype, by only reading from the city buses' CAN network without writing to it; and d) they are able to detect faults, assign health statuses to each vehicle, and diagnose faults when deviations occur, by using the vehicle service record database's repair dates.

2.15.2.4 Drawbacks

The drawbacks are as follows: a) they do not provide a system architecture of their prototype, any details on the hardware used to capture the CAN network data, or any details on data exploration; and b) they are not clear how they define the threshold for determining deviation.

2.15.3 Approach 3 - Self-Organizing Maps for Automatic Fault Detection in a Vehicle Cooling System

[87] propose a hybrid predictive maintenance approach by combining a data-driven method and expert systems. They use expert knowledge to select key sensor signals for detecting a specific sub component's faults, and a data-driven approach to analyze sensor data to detect deviations. They attempt to predict cooling system faults in a fleet of city buses by choosing interesting signals relevant to the cooling system, and performing fleet-wide analytics on such signals. They choose the signals that are most significant to the state of a cooling system, and send models of theses sensors to a server by using a wireless network connection.

2.15.3.1 Deviation Detection

They use the Self-Organizing Map (SOM) clustering method, which is a neural network, to assign vehicles into fault groups. They use the Euclidean distance between all the nodes in the SOM models to assign vehicles to various fault groups. The vehicles' SOM distances that deviate from the rest of the fleet indicate a faulty cooling system. Each vehicle has a SOM model to abstract the vehicle sensor data. All the vehicles have the same number of nodes and the same topology in their SOM models. To compare two vehicles, they compute the Euclidean distance between all the nodes of both SOM models denoted by A and B. The distance between all the nodes of A and B is denoted as d_{ij}^{AB}. They pair the nodes of the SOM models such that the sum of their distances is minimized, which is used as the distance between the models, denoted as $D_{A,B}$. They do this pairing using a random search. They use an assignment matrix S^{AB}, such that the elements are either 0 or 1, where 1 indicates nodes are connected and 0 indicates they are not. In other words, $s_{ij}^{AB} = 0$ indicates that node i in A is not connected to node j in B, and $s_{ij}^{AB} = 1$ indicates they are connected. A row can have at most one 1 inside it, the rest of the elements must be 0. The distance between A and B can therefore be represented as follows in equation 2.15:

$$D(S^{AB}) = \sum_{i=1}^{N} \sum_{j=1}^{N} s_{ij}^{AB} d_{ij}^{AB} \tag{2.15}$$

They then define the distance between A and B as the minimum of all such distances $D_{A,B}$ =min$(D(S^{AB}))$. Since the assignment matrix can have many possible configurations for representing how the nodes are connected, $N!$, they approximate the minimum distance $D_{A,B}$ using an iterative random search.

2.15.3.2 Experimental Data

To illustrate their proposed method, they inject engine cooling system faults. They gather data from one city bus, which was driven for 35 minutes in the city on various terrain types. They inject three different radiator clogging levels, namely, 0%, 25%, and 50%. The sensor signals they analyze are as follows:

- Engine Fan Speed
- Turbo Pressure
- Coolant Temperature
- Engine Speed
- Engine Load
- Oil Temperature

- Pedal Positions
- Road Speed

2.15.3.3 Results

Their data analysis demonstrated a difference in behavior between varying clogging levels of radiators.

2.15.3.4 Advantages

The advantages are as follows: a) there are fewer signals that need to be modeled, since their approach attempts to specifically detect when a specific subsystem fails the cooling system; b) less wireless bandwidth is used and fewer computations are required; c) fleet-wide analytics are used; and d) they had access to fault data; that is, the level of clogging in radiators by injecting faults into the test bus.

2.15.3.5 Drawbacks

The drawbacks are as follows: a) the sensors chosen are not selected automatically, since they require expert knowledge to choose the sensors that would most likely best represent the state of the cooling system; b) they do not give any details on the system architecture they use to acquire the data, send it to a server, and process it on the server.

2.15.4 Approach 4 - Consensus Self-organized Models Approach — COSMO

[82] proposes a data-driven predictive maintenance approach for a bus fleet. Their system can be separated into three phases: sensor selection, deviation detection, and fault diagnosis. Their goal was to detect which bus has or will break down, and once their system detects a fault, it attempts to diagnose the fault. Their main assumption is that the majority of the vehicles are healthy in their fleet. Some of the older work of some of the authors is found in [18], [79], [81], [71], [28], [29], and [87].

They focus on working with less critical equipment (that is, affordable equipment that can/will break) to learn and acquire knowledge after the system goes into production. This design choice is inspired by the fact that fault data is difficult to obtain, and in most situations equipment owners cannot afford to let expensive equipment break-down. They equip a fleet of city buses with gateways. Their gateways read sensor data from the on-board network, and transmits models/aggregates of the data, via telematics, to a back-end server for fleet-wide analytics. Instead of sending the raw signals over the network, they model the signals using three models to capture the on-board data streams: histograms, autoencoders, and linear relations between pair-wise signals. They read sensor signals from

the vehicle ECUs. They can not include all signals into their machine learning models, which means they have to choose the most useful sensors to include in their models. Their cloud server performs fleet-wide vehicle anomaly detection, and by data mining a vehicle service record database (VSRDB) it attempts to diagnoses the deviations.

2.15.4.1 Sensor Selection

They decide to choose sensors based on interestingness, which they define as signals that are not **random** and are **stable** between consecutive days. They define random as a signal that has many different values (high cardinality) and non-random as a signal that produces fewer different values (low cardinality). For example, a signal that produces only two values is less random, and therefore more interesting, than a signal that produces thousands of different values. Their logic is as follows: when a generally non-random signal deviates from the rest of the fleet, it is clear that a deviation has occurred. However, if a signal is random, then it is not clear whether a deviation actually occurred or whether the normal sensor signal's behaviour deviated naturally by chance.

They also define an interesting signal as a sensor that is stable or consistent between consecutive sampling periods; that is, a sensor signal is stable if during two daily consecutive daily samplings, the model for the first day and the model for the second day are similar. Conversely, a non-stable signal would produce two daily consecutive models with great differences between each other.

2.15.4.2 Deviation Detection

The vehicles gather sensor data, choose their most interesting signals, and report their models back to their cloud server. The cloud server identifies vehicles that are different from the rest of the fleet. They detect deviations between vehicle models, requiring only the ability to compute the distance between models. The null hypothesis is that all the sensors of the same type output the same distribution for all vehicles. With the null hypothesis in mind, using the distance between models, they compute a z-score (a deviation score), using the Most Central Pattern (MCP) method, for all the vehicles' models. The z-score represents the probability the vehicle's model is deviating from the rest of the fleet. They take a 30-day moving average of z-scores, named a z-score, which they use to check against a threshold for deviations.

Most Central Pattern

To detect deviations, they compare a test bus to the rest of the fleet, which requires a distance metric between models. They collect the models from each bus for a week and produced a symmetric distance matrix that has the pair-wise distances between models.

$$D = \begin{bmatrix} d_{11} & d_{12} & d_{13} & \dots & d_{1N} \\ d_{21} & d_{22} & d_{23} & \dots & d_{2N} \\ \dots & \dots & \dots & \dots & \dots \\ d_{N1} & d_{N2} & d_{N3} & \dots & d_{NN} \end{bmatrix} \tag{2.16}$$

Where $d_{ij} = d_{ji}$ for distances in D, and d_{ij} denotes distance between model i and model j. The matrix has to be sufficiently large (have many models), to enable the detection of normal and abnormal models. So either a) for a big fleet each vehicle only includes a single model in the distance matrix, or b) for smaller fleets each day for a short period, each vehicle has their model placed in the distance matrix. The approach is effective when it has multiple models either of the same vehicle or different vehicles. To compute the z-score, deviation score, for some test vehicle's model m, where t is the test vehicle, the MCP method does the following:

1. Remove all distances related to t from D. Let $i \in t$ denote that model i is a model from the vehicle t. Therefore, by removing d_{ij} from $D \ \forall \ i \in t$ and $j \in t$, we obtain the submatrix D'.

2. Find the most central model, c, in D'; that is, the model that is nearest to the mean of the models, which is the model whose row has the minimum sum, since it has the least total distance to all models in the matrix, where $c = \min(\sum_{j=1}^{N} d_{ij}), \forall i = \{1, ..., N\}$.

3. Compute the deviation score, the z-score, which is estimated as the ratio of models that lie further away from c than the distance from m to c (see equation 2.17), where N_s is the number of rows in D', and d_{ic} is the distance of model i to the most central model c. The model c is not included in the distribution when calculating the z-score for model m.

$$z - score = \frac{|\{i=1,...,N_s \ : d_{ic} > d_{mc}\}|}{N_s} \tag{2.17}$$

The MCP is built such that if the Euclidean distance was chosen as the model's distance measures, then the most central model will be the one that has the row with minimum sum. Using the z-score, they assign a p-value, which will be used for detecting deviations by computing the moving average of the z-scores for each signal. To detect deviations, the moving average is compared to a pre-defined threshold. This is statistically correct, since they assume the distributions of the models are the same distribution, so z-scores are uniformly distributed between 0 and 1.

2.15.4.3 Fault Diagnosis

They diagnose the deviations using a supervised machine learning approach, which requires a large knowledge base of labeled fault data. When performing vehicle diagnosis, they need to link deviations to repair dates. Since their VSRDB was ill formed, they link deviations to repairs with a period of 4 weeks, 1 week before and 3 weeks after the deviation goes below the threshold.

They predict the diagnosis required to fix faults with buses, by suggesting what component most likely needs replacing. They accomplish this using the VSRDB, by analyzing the proprietary Volvo repair codes. They predict the expected number of fault codes for

a given four-week period and compare it to the actual number of fault codes found. They analyze the expected frequency of repair code occurrences in the VSRDB and compare those results to the observed frequency of repair codes. They calculate the expected frequency by selecting a bus and a random 4 week period and then counting the occurrences of each repair code. They do this sampling for 33 time periods, and obtain the occurrence counts of the observed counts in their period of interest. They do this 50,000 times to get the mean of the occurrences, to approximate the expected number repair code occurrences and the standard deviation σ. They estimate the repairs required to fix a deviation by identifying the most common repair codes found during the deviation period.

2.15.4.4 Models

Histograms

They use histograms for their robustness. Histograms cannot represent time-series data. Bin widths can automatically be created on-board the vehicles, since by using the J1939 specification document, or CAN in their case, a database of sensor specifications is available, which defines the maximum and minimum values for the sensor.

Histograms - Interestingness

They use entropy, which is used in section 2.12.6 to define interestingness for histograms. They define the entropy of histogram $P = (P_1, ..., P_N)$ as follows in equation 2.18, where P_i is the normalised frequency of the data in bin i:

$$E = -\sum_{i=1}^{N} P_i log(P_i) \tag{2.18}$$

where the normalized frequency is the frequency of a bucket divided by the total frequency of each bucket; that is, normalized frequency of a bucket can be defined as follows in equation 2.19, where B_i is the frequency of a bucket i, and N is the number of buckets:

$$P_i = B_i / \sum_{j=1}^{N} B_j \tag{2.19}$$

To compare two histograms (two different signals), they use normalized entropy, which is defined below in equation 2.20. It ignores empty bins and allows histograms with different number of bins to be compared. N is the number of non-empty bins instead of the number of bins. Lower NE values indicate uninteresting signals, since they have high entropy, and higher values are more interesting, since they indicate a concentration of values.

$$NE = \frac{log(N) - E}{log(N)} = 1 + \frac{1}{log(N)} \sum_{i=1}^{N} P_i log(P_i) \tag{2.20}$$

Histograms - Stability

To measure the stability of a signal, they compare two histograms of two consecutive days. To compare the histograms, they use the Hellinger distance to find the variation of the histograms. The advantage of Hellinger distance is its ability to deal with empty bins. They assume comparing two histograms should be possible, since the signals come from

the same data distribution. The Hellinger distance can be defined as follows in equation 2.21, where P_i and Q_I are bin values for normalized frequency (probability) histograms P and Q, respectively, and $0 \leq H(P,Q) \leq 1$.

$$H(P,Q) = \sqrt{\frac{1}{2}\sum_{i=1}^{N}\left(\sqrt{P_i} - \sqrt{Q_i}\right)^2} \tag{2.21}$$

Autoencoders

Autoencoders can represent the characteristics of the time series nature of a signal. The autoencoder is an artificial neural network that is trained to output the input it receives. They split the time series data into windows, providing the windows as input to the autoencoder. They compress the input data $x \in [0,1]^d$ into a hidden representation $y \in [0,1]^{d'}$, using deterministic mapping:

$$y = s(W \cdot x + b) \tag{2.22}$$

where s is a non-linear function, W is a weight matrix, and $d' < d$. Using this mapping, the output $\hat{x}$ is reconstructed and should be as similar as possible to the input data it was trained with.

$$\hat{x} = s(W' \cdot y + b') \tag{2.23}$$

They attempt to reduce the reconstruction error by using cross-entropy:

$$L_H(x,\hat{x}) = -\sum_{k=1}^{d}\left[x_k \cdot log\hat{x}_k + (1-x_k)\cdot log(1-\hat{x}_k)\right] \tag{2.24}$$

This way, when the error is small, the input provided is from the same distribution as the training input data. The error will grow when the input data is not from the same distribution. They also introduced noise to make their model more robust. Let x_r denote the reference data that will be used to compare the distance between two autoencoders, a and b, which were trained using the data x_a and x_b, respectively. They calculate $[z_{ar}, z_{br}]$, where z_{ij} is the input reconstructed by autoencoder i and data x_j. They use the Normalise Mean Square Error (NMSE) for comparing outputs when given x_r as input.

$$D_{ab} = NMSE_{ab} = \frac{1}{N\sigma^2}\sum_{n=1}^{N}\left[z_{ar} - z_{br}\right]^2 \tag{2.25}$$

where σ is the standard deviation. Autoencoders a and b that were trained on the same data should produce similar results when presented with the same input, so D_{ab} should be small, while if a and b were trained on different data distributions, the error D_{ab} should be large.

Autoencoders - Interestingness

The randomness of the signal is captured by the reconstruction error (see equation 2.24), instead of entropy in the case of the histogram. The stability is also calculated by measuring the distance between two consecutive samples (see equation 2.25).

Linear Relation

They use linear relations to take advantage of the physical relationships between signals. They take all pairwise linear models between two distinct signals x_i and x_j of the form

$$\hat{x}_i = a_{ij}x_j + b_{ij} \tag{2.26}$$

Linear Relation - Interestingness

A pair of signals x_i and x_j are interesting if the linear relationship is accurately represented by equation 2.26. This can be computed as follows using the NMSE:

$$NMSE_{ij}^v = \frac{1}{N\sigma^2} \sum_{n=1}^{N} [x_{ij}^v(n) - \hat{x}_{ij}^v(n)]^2, \tag{2.27}$$

where σ is the standard deviation. The accuracy is computed for the entire fleet.

$$\alpha_{ij} = \frac{1}{V} \sum_{v=1}^{V} NMSE_{ij}^v \tag{2.28}$$

Linear relations have a second aspect of interestingness, β_{ij}, which represents how much the parameters vary across the fleet.

$$\begin{aligned} d_{ij}^v &= max_{w=1..V}\left(\sqrt{(a_{ij}^v - a_{ij}^w)^2 + (b_{ij}^v - b_{ij}^w)^2}\right) \\ \beta_{ij} &= \sqrt{\frac{1}{V} \sum_{v=1}^{V} (d_{ij}^v - \bar{d})^2} \\ \bar{d} &= \frac{1}{VKK} \sum_{v=1}^{V} \sum_{i=1}^{K} \sum_{j=1}^{K} d_{ij}^v \end{aligned} \tag{2.29}$$

Interesting pairwise signals are those with small α_{ij} (strong relationships) and large β_{ij} (large variations between vehicles), since it would suggest a fault occurred. In this case a low value of β_{ij} would suggest no fault occurred yet. Therefore, these relationships should be recomputed periodically to find faults.

The Euclidean distance between paired signals is used to populate the distance matrix.

$$D_{vw} = \sqrt{(a_{ij}^v - a_{ij}^w)^2 + (b_{ij}^v - b_{ij}^w)^2} \tag{2.30}$$

2.15.4.5 Experimental Data

They collected models daily for a week, for a fleet of 19 buses. Their experiments were conducted in Sweden, and involved a fleet of 19 public transport buses. They drop error values in their experiments.

They acquire data over a J1587 network and two CAN networks. They use a GPS signal to track buses. They use a gateway device named Volva Analysis and Communication Tool, connecting the vehicle to the internet wirelessly. They are able to remotely administer their gateway. Their gateway was read-only on the vehicle networks. They determine when buses were in the garage using GPS signals and the garage coordinates.

2.15.4.6 Result

They found the following : a) a runaway coolant fan that the mechanics could not detect, b) NO_x sensors that were failing, c) a jammed cylinder, and d) a malfunctioning wheel speed sensor.

2.15.4.7 Advantages

The advantages of their work follows: a) COSMO is resistant to seasonal changes over time, since it compares all the vehicles of the fleet to each other; b) it works well with a fleet of similar equipment; c) the COSMO approach does not require expert knowledge; d) their approach supports on-board and fleet-wide analytics, which makes resource management feasible; e) their approach is flexible, since a variety of machine learning models can be used to model the data; f) the sensor/signal selection (choice of model) can be performed almost entirely on-board the vehicle, which saves network bandwidth; and g) it can be deployed with low development time and supports automation, since their approach does not use expert knowledge (it is unsupervised).

2.15.4.8 Drawbacks

The drawbacks of their work are as follows: a) They use an ill-formatted VSRDB. b) In their results, using the autoencoder model, there were repairs made when no deviations were detected. They suspect the repairs were done too early and believe it was not a no-fault-found (NFF) occurrence. On another occasion they mention that a repair may not have been linked to an actual fault. This suggests it is not clear if some faults actually occurred, which means a deviation may not be linked to an actual fault, or a NFF may have occurred. c) Their sensor selection method is not guaranteed to select optimal sensors for fault detection. d) Their diagnosis data mining approach requires Volvo repair codes, which are proprietary. Therefore, it would be difficult to replicate their diagnosis work. e) They do not provide a detailed architecture of their IoT system put in place to gather the data and they claim to gather data on a USB stick. This does not appear to be a robust IoT system prototype.

2.15.5 Approach 5 - MineFleet

A literature review of [49] and [48] reveals that MineFleet is one of the first commercial predictive maintenance systems. MineFleet is a distributed stream mining system and has an IoT architecture for fleet management, focusing on distributed stream management. Each vehicle is equipped with a gateway, which taps into an OBD-II network for reading vehicle diagnostic information. Each gateway performs on-board analytics, such as correlations, inner-products, and Euclidean distance matrices. The MineFleet Real-Time system uses the Fast Monitoring of Correlation matrix algorithm. By performing on-board analytics, MineFleet can send the resulting aggregations of the sensor data to the cloud (via a wireless network connection), instead of sending the raw signals. It also provides an application web interface, which illustrates the gathered data findings for managers. MineFleet is used for modeling, benchmark evaluation, vehicle health monitoring, driver monitoring, fuel-consumption analysis, and fleet characteristics mining. Examples of fleets that can benefit from MineFleet are grocery vehicles, airplanes, and spaceships.

MineFleet focuses on decentralized distributed mining on mobile data sources; that is, on mobile agents. MineFleet manages on-board data stream mining using fog devices, and it keeps the following resource constraints into consideration: minimizing wireless network bandwidth, data storage, and computational resources. The goal is to mine streams of data quickly and efficiently, while keeping security in mind.

MineFleet offers web services that report the results of the fleet-wide data analysis. They use a relational database to store the fleet data. MineFleet performs health monitoring by monitoring various subsystems. It conducts health tests on each vehicle, and compares the health test results of the entire fleet to find an unhealthy vehicle. With the health test results, MineFleet also offers fleet managers multiple options to improving fleet performance. For example, MineFleet presents the top four best options to improve fuel efficiency to fleet managers.

2.15.5.1 Advantages

MineFleet supports many IoT and fleet management requirements: a) supports on-board and fleet-wide analytics; b) provides visualization tools for the discovered knowledge; c) provides distributed applications support, due to the nature of its decentralized data mining; d) enables resource management; e) supports mobile agents; f) proposes fleet optimization strategies to fleet managers; g) detects vehicles that deviate from the rest of the fleet; h) provides the details of the hardware required to deploy a predictive maintenance system using MineFleet, i) and it considers security in its design.

2.15.5.2 Drawbacks

The architecture of MineFleet is not completely distributed, since it only has one back-end server, instead of having a massive network of machines that support the fleet. MineFleet is a commercial product and does not appear to be open source. The assumptions made

by MineFleet about wireless technologies are out-dated. [49] and [48] mention that prior to 2006, MineFleet customers were not ready to pay for a data plan that was over 5MB per month. Since modern companies can afford wireless data plans that exceed 100GB per month in 2019, this suggests MineFleet is an ancient technology and that its assumptions and design may not be compatible with modern software and hardware.

2.15.6 Approach 6 Cloud-based Driver Monitoring and Vehicle Diagnostic with OBD2 Telematics

[3] propose an expert system-based predictive maintenance approach. They read data from an OBD2 port, process it using a set of user-defined rules, and send events and sensor data aggregates to a server. They perform driver behavior monitoring, but this is outside the scope of this thesis, so I will not cover the details of it here. They present key information to the users of their Android application and via a web server.

2.15.6.1 Predictive Maintenance System Architecture

They perform vehicle diagnostics using OBD-II, an Android app, and a cloud server. Their Android application gathers sensor data from OBD-II using an OBD-II Bluetooth adapter named ELM-327, and it performs predictive analytics and provides data storage and visualization. Their cloud server is used for storing long histories of data, which are stored using NoSQL. They use event processing both in the cloud (on their server) and in the fog (on their Android device). The Android application sends sensor data, using HTTP, to the back-end server over a 3G or 4G network. They perform real-time queries on the sensor data to detect events; for example, high fuel consumption. Users can add queries dynamically to their rule-base, which adds flexibility to their system. More complex event processing is performed in the cloud. They use WSO_2 BAM, which supports Apache Cassandra integration.

2.15.6.2 Cloud-based Sensor Failure Detection

They propose a solution to detect sensors that are gradually failing. The sensors they focus on are processed by the back-end server and include the O_2 and Mass Air Flow (MAF) sensors. They use a rule engine to decide whether the O_2 sensor is failing, by applying expert knowledge to define threshold values. Their Android application reads the maximum and minimum values of the O_2 sensor for 15 min when the vehicle starts, and then the Android application sends the readings to BAM. The back-end server then uses regression on the maximum and minimum values and attempts to predict the potential date of failure. To predict when the MAF sensor will fail they analyze the linear relationship between the MAF sensor readings and the engine RPM. When a MAF sensor begins to fail, the gradient will begin to reduce gradually. They use regression to predict the time remaining before the sensor fails.

2.15.6.3 On-board Analytics

They attempt to detect faults using an on-board rule engine on the Android device. For example, when the moving average of the engine oil temperature exceeds 104°C, a rule is fired. The sensors included in the on-board rule engine are as follows: engine oil temperature, engine RPM, engine coolant temperature, fuel consumption, speed, and battery voltage.

2.15.6.4 Experimental Data

They do not have enough data to run full experiments to produce results that would illustrate their system's functionality. Therefore, they generate synthetic data to run these experiments, which are complemented with some live data in the form of maximum and minimum voltage values of the O_2 sensor. They also do not have access to fault data, so they could not test their fault diagnostic algorithms using real data.

2.15.6.5 Results

They implement both an Android application that supports their predictive maintenance system and a web application that reports results obtained from the back-end server. Using their synthetic data used to simulate faults, they are able to predict the remaining useful life of the O_2 and MAF sensor.

2.15.6.6 Advantages

The advantages are as follows: a) since an Android device acts as the gateway and a lot of smart phone development has been done in recent years, deploying the gateway and predictive maintenance applications will require a relatively short development time; b) their architecture supports both on-board and fleet-wide analytics; c) they predict the remaining useful life of sensors using synthetic and live data; d) their predictive maintenance system can adapt to environmental changes, since users can dynamically add rules to the on-board and fleet-wide rule engines; and e) they provide a good explanation on the hardware required to read from a OBD-II network, how they design their architecture, and how they implement their rule-engines and other predictive analytic algorithms.

2.15.6.7 Drawbacks

Their system is not fully autonomous, because of the following: a) an expert needs to select the sensors of interest, b) an expert needs to define rules for the rule-engines, and c) to deploy the gateway on a vehicle, a smart phone is required by a passenger, which is less autonomous than a gateway that is physically installed onto the vehicle. For the M2M communication between their devices, they use HTTP as the application layer networking protocol instead of a lightweight M2M protocol such as MQTT or CoAp. In a system

with millions of sensors and devices, the overhead of processing millions of HTTP GET requests is not feasible. Their system will have difficulty detecting new faults, since it uses an expert knowledge-based approach and the system can only be as accurate as its rules defined by experts. Their architecture supports fleet-wide analytics, but their proposed algorithms do not perform fleet-wide analytics. They only analyze sensor data of vehicles individually.

2.15.7 Approach 7 - An IoT Gateway Centric Architecture to Provide Novel M2M Services

[21] proposes an IoT architecture designed to connect legacy devices to the internet. They separate their architecture into three layers: the sensing layer, the gateway API layer, and the application layer. They connect legacy machine-to-machine (M2M) devices, for example sensors and actuators, to the internet. To do so, they equip the legacy M2M devices with logic to publish their data readings to the cloud over HTTP, where the data is encoded in SenML using JSON. These readings are sent from the sensing layer to the gateway API layer, which hosts the wireless gateway.

The wireless gateway connects all the legacy devices together, to provide applications with an API to access the sensors and actuator via a mobile phone application. Some legacy devices do not have the wireless networking capabilities to connect to the wireless gateway, so additional (smaller) wireless gateways are deployed (which they call intermediate sensor gateway) to bridge non-wireless legacy devices to the wireless gateway. The wireless gateway connects M2M devices to the entire system, enabling application clients to access the M2M devices. The gateway is responsible for discovering the M2M devices. Note that some of the M2M devices are connected to the gateway via a serial communication protocol Modbus. The clients can dynamically remove any of the M2M devices connected to the gateway in real-time.

They use SenML, which is encoded using JSON, for encoding and manipulating sensor data. SenML is typically used only for encoding sensor readings, but they expand SenML to offer actuator manipulation as well. They expand SenML by adding a new resource type description, which enables actuator and sensor packets to be differentiated. To then configure an actuator, they also add a variety of parameters to SenML packets, such as the list of supported actuator functionalities.

They create a prototype to showcase their architecture, which uses the Google App Engine and RESTful web services API. Their prototype includes an Android tablet as the application client of the system, which reads sensors values and activates actuators. The application servers that interact with the clients are responsible for connecting clients to the endpoint devices. To discover the end point devices requested by a client, the application server queries the gateway's local database for connected endpoint devices. As devices connect and disconnect from the gateways, clients that have subscribed to those devices receive a notification. The client can also subscribe to endpoint devices of interest to receive sensor data readings.

2.15.7.1 Advantages

Their architecture is distributed, which means it will scale well with an increasing number of devices. Since all their M2M devices (sensors, actuators, and intermediate sensor gateways) communicate with the wireless gateway, their architecture supports mobile agents, because the wireless devices may physically move around and maintain their connection to the system. Their architecture's gateways host databases to store device metadata, which increases locality and therefore availability. Their architecture allows legacy devices to connect to the internet. They provide many details on the application layer of their prototype, such as the APIs used in the prototype, the programming language used, the application protocols, and sequence diagrams to properly illustrate their design.

2.15.7.2 Drawbacks

There is a lot of unnecessary overhead in their architecture:

- The SenML packet definition used creates overhead. That is, when they encode a sensor reading into a SenML packet, the packet also includes a time stamp, sensor name, and multiple other parameters. This is not a scalable design, since they are adding a lot of unnecessary bytes to SenML packets. With only a sensor reading, its time stamp, and the sensor name in a packet, the packet size is increased by a factor of 3. Instead, multiple readings could be sent in one packet, which would reduce the overhead, and instead of adding a timestamp to each sensor reading, timestamps should only be included with aggregations of the sensor readings.
- Regarding the use of HTTP, this is the same drawback as the HTTP drawback in section 2.15.6.7.
- Their architecture and other proposed protocols fail to implement any form of sensor data aggregation, which should be used to reduce network utilization. Instead they send raw data streams over the network to the wireless gateway.

They do not mention what type of M2M device they use. It is not clear if they use simulated or real sensors and actuators to produce the results discussed in their work. They do not describe what hardware they use as the gateway.

2.15.8 Approach 8 - Connected Vehicle Diagnostics

[104] propose an expert-system solution for connected vehicle diagnostics and prognostics to address the following issues: there is a limited amount of available vehicle sensor data, and experts can easily be overwhelmed when analyzing the data of an entire fleet. Diagnostic algorithms are used to filter the massive amount of data to a manageable size for an expert. Experts are intended to use the results of such algorithms to help them diagnose faults; the algorithms are not replacing the mechanics.

They perform a study with General Motors (GM), which focuses on the battery of the vehicle during the design validation process, before launching a new vehicle into the market. They focus on the battery, since a non-functioning battery means the vehicle cannot start, and they focus on the battery's inadvertent load, since it is a good indicator of vehicle faults. They propose an algorithm to monitor this parameter to try to predict faults. Periodically the battery management ECU would read the battery's current and the current the ECU is drawing. They chose to avoid continuous battery monitoring, since having the battery management system ECU always online would drain the battery.

They also propose a voltage-based fault diagnosis algorithm and a decision-support system based on a rule engine. For each rule, they associate a confidence level between 0 and 1. A confidence level indicates the level of certainty on has that a rule detects an anomaly. This is used because it is normal for an abnormal behavior to occur periodically, and such behavior should not be considered as a fault. For example, for a vehicle that has been parked for a long time, when ignition occurs, it will not have the same effect on the battery as a car that is started regularly. In their system they have domain experts define the confidence levels for each rule. When rules fire, they are grouped by their corresponding anomalies, and each anomaly has its confidence computed by adding all confidence levels of the rules in the groups. To combat false alarms, visualization tools were used by the mechanics with domain knowledge to determine if a fault was a false alarm. The anomalies were then sorted by confidence level and visualized to bring potential anomalies to the attention of the data analysts, who also add new rules. When potential anomalies are displayed to analysts, they can further investigate the anomaly by viewing the behavior of sensors over time, which can be done by selecting the desired sensor found in the fired rules.

They conduct a proof of concept study, by equipping a fleet of 20 test vehicles with ECUs that have wireless communication capabilities. These ECUs upload battery parameters to a central database. To test their system, they inject faults into a select set of test vehicles and invite data analysts to use their system to see what they could find. The data analysts were not told about the injected fault, since they were testing the usability of the proposed system. One of the fault-injected vehicles started normally, providing no visual indications of the underlying battery issue found on the vehicle. What is interesting is that the analysts were still able to detect the battery issue on this vehicle by using the proposed decision-support system. The system was later deployed into production using OnStar.

2.15.8.1 Advantages

Their proposed decision-support system evolves over time, since the mechanics have the ability to add new rules to the systems. The decision-support system allows data analysts to investigate potential anomalies that were not detected by the system. Their system demonstrates a quick and cost-effective way to detect infrequent faults. By analyzing fleet-wide data instead of a single vehicle's data, their system narrows and reduces troubleshooting time by narrowing the source of observed faults. They manage the vehicle's battery by periodically powering the ECU that manages the sensor responsible for reading battery parameters.

2.15.8.2 Drawbacks

Since they only periodically monitor the battery current, they miss current readings when the battery management ECU is not powered on. They do not have any on-board analytics implemented in their system; that is, they acquire battery data (from an ECU with wireless communication abilities) and immediately upload it to their system's database. At no point do they mention that intelligence is applied on-board the vehicle before the data is sent to the database. The rule engine is only executed in the cloud. With a large fleet of vehicles, sending all this data may not be feasible without on-board analytics.

2.15.9 Approach 9 - Military Vehicle Diagnostics

[34] introduce a predictive maintenance approach using telemetry. In their approach, vehicle sensor data is acquired through vehicle diagnostic systems, such as OBD-II and J1939. Sensor data is sent to the cloud using wireless telemetry equipment. They conduct an experiment on military vehicles to gather data and associate the sensor behavior to the terrain the military vehicles are driving on. The goal of the experiments is to maximize the military vehicles' gearbox and hydro-retarder loads by analyzing the oil temperature in the gearbox and of the hydro-retarder. They acquire data from the vehicle's J1939 network, using the Vector CANcase logging device, and analyze the data using the CANoe software by Vector.

Their results reveal maximum sensor values from the hydro-retarder were found to occur when driving down hill. Some of the vehicles had clutches that already suffered some damage; that is, the clutches were burnt.

2.15.9.1 Advantages

They provide many design details of their proposed predictive maintenance architecture. To demonstrate a live use case of their architecture, they create a prototype that enables data acquisition of real data. Their prototype provides automated data acquisition using wireless technologies, since they do not manually acquire the data using a USB stick. They analyze this data offline and provide many experimental results.

2.15.9.2 Drawbacks

They do not propose any machine learning or artificial intelligence approaches that attempt to predict the health status or remaining useful life of military vehicles. Instead, they only use a pre-existing software tool, CANoe of the firm Vector, to analyze the data for them. That is, they simply analyze the data trends without proposing any prediction models.

2.15.10 Approach 10 - An IoT Gateway Middleware for Interoperability in SDN Managed Internet of Things

[15] proposes an IoT middleware for managing IoT devices, using gateways and software-defined networks. Although a software-defined network is beyond the scope of this thesis, their IoT architecture is quite similar to the one proposed in this work, and therefore, their architecture's design is included in this section. In their architecture, the IoT gateway connects resource-constrained devices, such as sensors, ECU's, and other edge devices, to the cloud. Their goal is to bring intelligence closer the edge of the network. They choose SQLite as the mechanism for storing IoT data on edge devices, since it does not require a SQL server. They chose the Raspberry Pi 3 as their IoT gateway. Since their design is flexible, instead of a Raspberry Pi, a more heavy-duty machine could be used as the IoT gateway.

They perform an experiment to demonstrate a prototype of their architecture. The hardware they use is as follows: a) a JY901 sensor, which is used to monitor the movement of objects, b) a Raspberry Pi 3, the IoT gateway, c) an iPhone 7, which is used as a Bluetooth device, and d) a BlackBerry Priv, which is also used as a Bluetooth device. They use the gateway to translate the wired and wireless protocols, in order to connect them to their system. The gateway connects these devices to their system by scanning all its interfaces: USB (for serial connections), Bluetooth, and Wi-Fi interfaces. They host their middleware layer in the cloud by using Microsoft Azure, which hosts many cloud services.

They evaluate the performance of their system as follows:

a) By analyzing the CPU usage (power, load, and speed), with a variety of tools such as `sysbench`, `mpstat`, `mbw`, and `PowerTop`.

b) When using various subscribe-publish protocols, namely: MQTT and AMQP.

c) Investigating the network performance by analyzing the round trip time between various nodes in their network, which can be achieved using the `ping` tool.

d) By analyzing the time required for an application to be notified of the removal or addition of a node on their network.

2.15.10.1 Advantages

Their architecture is flexible, and supports various protocols to enable interoperability between devices. Intelligence can be deployed at various positions in their architecture, allowing intelligence to be found either on the edge of the network or in the cloud, which reduces network traffic. Their architecture also supports light-weight client-side protocols such as MQTT, CoAp, and AMQP, which are ideal for resource-constrained IoT devices. They use an embedded database, SQLite, instead of a full-fledged DBMS, for storing IoT data. They give detailed technical information on the hardware used in their experiments. They perform an evaluation of their prototype using various metrics.

2.15.10.2 Drawbacks

They do not test their architecture in a live environment. They do demonstrate a use case of their architecture, but they do not demonstrate their architecture in the context of an IoT application such as smart transportation, smart health, or smart cities. They only demonstrate experimental performance results from their experimental setup.

2.15.11 System Requirement

The literature review conducted in chapter 2 reveals that there are two sets of important requirements for a predictive maintenance system: IoT middleware/architecture requirements and fleet management system requirements. A literature review of [52][15] reveals the following **IoT middleware** requirements: scalability, edge/fog computing, efficient IoT communication, multi-protocol support, cognitive computing, support diverse set of distributed applications, resource management, geographical distribution, connect networks of sensors, and data multi-tenancy. Chapter 2 reveals important **fleet management** requirements, which include mobile agents support, sensor/actuator accessibility, non-intrusive deployment (the system should not interfere with current fleet operations), visualize discovered knowledge, etc.

Below are the list of compiled requirements used to compare the approaches discussed in section 2.15. The comparison of approaches can be found in Table 2.4.

1. Provides a use case of the architecture

   A use case complements an architecture's design by proving it works in practice
2. System architecture details provided
3. Supports scalability

   The architecture should be able to handle many devices and a lot of data
4. Enables geographical distribution
5. Mobile-agent support

   Supports nodes that physically travel/move
6. Enables fleet-wide analytics

   Central data analytics of the fleet's data should be possible
7. Enables on-board analytics

   Edge/fog computing

   Analytics on-board the vehicle should be possible
8. Data analytics involved live data

   Data analysis with live data is more insightful than synthetic data

9. Fully autonomous
    - Human intervention should be minimized
    - Supports non-intrusive deployment
10. Efficient IoT communication
    - Uses a M2M protocol
    - Sends only aggregates to the cloud
    - Performs local computation when possible
11. Multi-protocol support
    - Supports access to various sensor network protocols
    - Supports various wireless networking protocols to connect to the internet
12. Enables connecting networks of sensors
    - Sensors and actuators should be accessible
13. Enables resource management
    - Power
    - Computations
    - Network Bandwidth
14. Enables data multi-tenancy
    - Access control to data should be easily partitioned
15. Support diverse set of distributed applications
    - Supports distributed-access API to IoT data and knowledge discoveries
    - Applications can access the distributed resources of the system
16. Knowledge discovery visualization
    - The system should support applications that can visualize the discoveries

Requirement		**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
Section	**Work**																
2.15.1	[81]	y	n	u	u	y	y	y	y	y	y	u	y	y	u	u	u
2.15.2	[18]	y	n	u	u	y	y	y	y	y	y	u	y	y	u	u	y
2.15.3	[87]	y	n	u	u	y	y	y	y	p	y	u	y	y	u	u	u
2.15.4	[82]	y	n	u	u	y	y	y	y	y	y	y	y	y	u	u	y
2.15.5	[49][48]	p	y	y	y	y	y	y	y	y	y	y	y	y	u	y	y
2.15.6	[3]	y	y	n	u	y	p	y	y	p	p	n	y	p	u	y	y
2.15.7	[21]	y	y	n	y	y	-	-	-	-	n	y	y	u	u	y	y
2.15.8	[104]	y	p	u	p	y	y	n	y	n	n	n	y	y	n	n	y
2.15.9	[34]	y	y	-	u	y	n	n	y	n	n	n	y	n	u	n	n
2.15.10	[15]	n	y	y	y	-	y	y	-	-	y	y	y	y	y	u	y
-	This Work	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	n

Table 2.4: Comparison of the Approaches Found in Section 2.15, using requirements defined in section 2.15.11. Where, "y" = yes (the requirement is met), "n" = No (the requirement is not met), "p" = partial (the requirement is somewhat met), "u" = unclear (the requirement may or may not have been met, it was not clear), and "-" = not applicable (the requirement is not relevant or applicable).

Chapter 3

My Approach - IoT-based Architecture of a Bus Fleet Management System

The present work proposes an IoT-based architecture designed to support fleet management for performing predictive maintenance. The architecture is illustrated in figure 3.1 and is inspired by the architecture proposed in [24]. [16] uses the architecture proposed in the present work and focuses on active rules designed for gateway deployment. It can be classified as a cloud-based middleware architecture (for more information on this type of classification see section 2.1.3). The architecture is divided into three layers: the perception layer, the middleware layer, and the application layer. All of these are variations of the IoT abstraction layers mentioned in section 2.1.1. The architecture contains three types of nodes, namely, vehicle nodes (VN), server leader nodes (SLN), and a root node (RN). Vehicle nodes represent the vehicles and are found in the perception layer, server leader nodes manage a region of vehicles and are found in the middleware layer, and the root node manages the entire fleet and is found in the application layer.

Although this work has defined only three types of nodes, the architecture is not limited to only these types. The architecture can be adapted to support additional types of nodes to meet the requirement of various deployment environments, since the three architecture layers mentioned above abstract their nodes' deployment details and represent the behavior of their nodes. For example, a smart sensor node could be added to the architecture. It would exist in the perception layer and represent a single sensor.

3.1 Meeting IoT and Fleet Management System Requirements

This section discusses how the architecture meets the IoT middleware and fleet management requirements, which are discussed in section 2.15.11. To meet the **scalability**

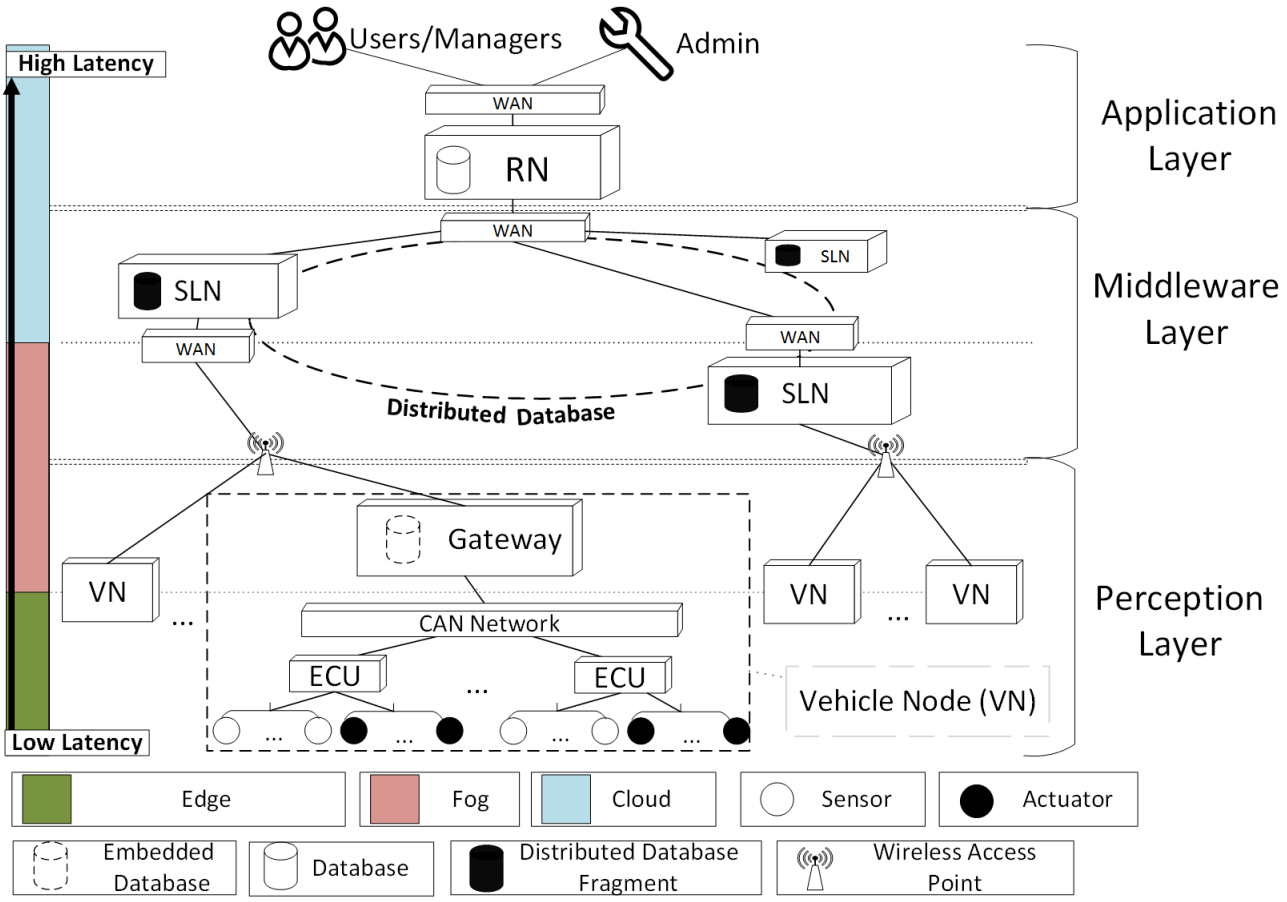

Figure 3.1: Predictive maintenance fleet management system architecture overview diagram. ECU = Electronic Control Unit, VN = Vehicle Node, SLN = Server Leader Node, RN = Root Node, WAN = Wide Area Network, CAN = Controller Area Network. Wireless Access Point includes Wi-Fi, LTE, GSM, Bluetooth, etc.

requirement, which also meets the **geographical distribution** requirement, the architecture is designed in a hierarchical manner. As a fleet grows, a fleet administrator can deploy additional SLNs (middleware nodes) in a geographical region to partition the region's VNs (perception layer nodes) into smaller sub-fleets to reduce the work load on an overloaded SLN. The **edge/fog computing** requirement is met by installing either a gateway onto a vehicle (in the fog) or a SLN in the fog. This type of fog computing also meets the **resource management** requirement, since the fog is the ideal location for managing resources. For example, the network communication can be reduced by making sure only aggregates of sensor data are sent over the network. Furthermore, perception layer nodes provide the ability to connect sensor networks together and to the cloud, which meets the **connecting networks of sensors** requirement. These nodes are also the best candidates for meeting the **data multi-tenancy** requirement, since they could be configured to manage sensor data and actuator access control. Middleware layer nodes can also partition sensor data access control to meet the data multi-tenancy requirements. To meet the **efficient IoT communication** requirement, the VN and SLN can aggregate data before transmitting

it over the network. Furthermore, MQTT is chosen for the communication between the perception layer and middleware layer, which also meets the efficient communication and resource management requirements. The architecture meets the **multi-protocol support** requirement, since there are a variety of protocols that can be implemented between the architecture's layers. For example, the interface between the middleware and the perception layer could be any of the following wireless protocols: Wi-Fi, Zigbee, Bluetooth, LTE, or GSM. Furthermore, the vehicle node can also support a variety of protocols for accessing sensor data and actuators. The **cognitive computing** requirement is met, since predictive analytics can be performed in the fog or the cloud. The **support diverse set of distributed applications** requirement is met, since the architecture is distributed and can therefore support a variety of distributed applications. To meet the **mobile agent** requirement, the VN is designed such that it has a persistent wireless network connection, LTE, for example, to access its SLN at all times. In fact, the **non-intrusive deployment** requirement can be met by enforcing read-only access to the CAN bus on the VN. Furthermore, the non-intrusive deployment is also met by removing the need for human users, which can be achieved using automation methods. The **visualize discovered knowledge** requirement is met, since an application can connect to the application layer to receive predictions and other knowledge that can be represented visually to a user using a web interface, for example.

3.2 Architecture Layers

The **perception layer** is the southern layer of the proposed architecture's layers, which is the closest layer to the data sources, and it abstracts both the fog (for more details on the fog see section 2.5) and the edge. Nodes in this layer do the following: a) interface to low-level edge nodes such as sensors, actuators, sensor networks, and embedded systems; b) perform sensing, lightweight storage, networking, and machine learning. These light weight functionalities are required in the fog, since naively sending all the sensor data to the cloud to perform predictive analytics would create huge delays [73][7], which defeats the real-time nature of the system; and c) connect to middleware layer nodes via wireless networking protocols such as Wi-Fi, LTE, GSM, and/or Bluetooth. MQTT is the protocol of choice used for communication between the perception layer and middleware layer, since it is lightweight on the client side. This design enables the support of many different types of heterogeneous perception layer nodes in the system. As long as they have the ability to communicate with middleware layer nodes via a wireless network connection using MQTT, they can be deployed in the fleet system.

The **middleware layer** is the layer between both the perception and application layers. It abstracts the fog and the cloud, which is important for an IoT system according to [15], since it allows its nodes to be deployed on either the cloud or the fog. Nodes in this layer generally perform more heavy-duty storage, networking, and machine learning compared to perception layer nodes. Both fog and cloud nodes are required, since the fog will not be replacing the cloud, but instead complements the cloud. This layer has two interfaces: a) the southern interface which connects (via MQTT) perception layer nodes to middleware

layer nodes, and b) the northern interface, which connects (via HTTP or other heavier protocols) the entire fleet system together by connecting all middleware layer nodes to the application layer node(s). In other words, this layer is the bridge between IoT devices and applications. Note that the MQTT broker(s), those enabling the MQTT communication for the southern interface, are found in this layer.

The **application layer** is the northern layer. It is similar to the IoT application layer mentioned in section 2.1.1.5. The node(s) in this layer host the main interface to the entire fleet system. The southern interface connects middleware layer nodes (sub-fleets) to the root node, and the northern interface connects application clients to the entire system via the root node. An advantage of this layer's configuration, according to [90], is that middleware layer nodes that connect to this layer's nodes can have a variety of configurations. That is, they can have a variety of different operating systems, resource requirements, and deployment environments, which means the proposed architecture is general enough to support a variety of heterogeneous middleware layer nodes.

3.3 Fleet Remote Administration

Each fleet node has a software administration agent/client installed, and this allows administrators to remotely update and configure the fleet nodes' software, via the administration server. The administration server is found in the application layer. Since a fleet can be large, it becomes infeasible to manage all the nodes in the network manually, which is why a software administration tool is required. With this in place, all other forms of node access can be disabled (for example SSH or serial shells), and as a result the only way to configure the fleet nodes is via the administration server, which is found on the root node (see section 3.6). Note that redundancy and crash recovery are outside the scope of this thesis, but in general they should be considered, since if the administration server fails access to the fleet is lost.

3.4 Vehicle Node

Found in the perception layer (see section 3.2), the vehicle node (VN) represents the vehicle. It hosts the *Fog-to-Thing* IoT architecture interface (these types of interfaces are discussed in section 2.1.3.1). The VN has a Controller Area Network (CAN) (such as J1939, which is described in section 2.9.3) and a gateway (such as a Raspberry Pi), which is described in section 2.3. By reading J1939 traffic, the gateway can perform sensor data acquisition, aggregation, and lightweight data analytics. The VN is the bridge that connects the edge nodes (ECUs in this case) to the middleware layer, which allows the other fleet nodes to access the vehicle's sensor data and actuators. The gateway can implement multiple software components to meet the requirements of the target deployment environment, which means it can be adapted to other predictive analytic applications. For example, in a smart health application, instead of tapping into a J1939 network, the gateway could be installed as a patch on a patient, and it could be connected to the patient's pacemaker

via Bluetooth. Figure 3.2 provides a SysML diagram of the software components involved with the VN.

3.4.1 Security

Security mechanisms can be used to secure the vehicle node; for example, configuring a firewall on the gateway. According to [7], the gateway should responsible for adding security mechanisms to lower level protocols that have little to no security. It is therefore the best candidate fleet node for adding a security layer above the CAN. For example, the gateway can be configured to host a software block that enforces read-only access to the CAN.

Since data tenancy is important for some IoT applications [52], the gateway is the best node to enforce this if multiple tenants own partitions of the sensor data. For example, if there are multiple CAN buses on a vehicle, where each CAN is owned by a different vehicle component manufacturing company, and the requirement is that each company wants to be able to remotely manage their equipment, companies should only be able to read and write to/from their designated CAN bus. The gateway could enforce security mechanisms to meet the CAN bus access rights requirement while offering each company access to their appropriate CAN bus.

3.4.2 Data Acquisition

The gateway can perform sensor data acquisition and can write to actuators, by tapping into the local CAN bus. When required, the gateway may store data locally to avoid sending data naively to the cloud, which achieves data locality and lowers latency. An embedded database should be used to store sensor data and other information, since it is lightweight and can be used as a single-user database, instead of dedicating a complete database management system (DBMS) on the resource-constrained gateway [52]. DBMS are good for systems with many users, but since the gateway has no human users (since it is only communicating with machines), a DBMS is not necessary.

3.4.3 Data Analytics

The VN offers on-board data analytics, by offering the ability to perform local computations on sensor data, and it enables fleet-wide analytics by sending data to its SLN when necessary. The VN itself is not in a position to perform fleet-wide analytics. The VN is in the best position to perform computations that only require the local sensor data, such as sensor selection, which filters unnecessary sensor streams and decoding sensor values. Instead of sending an entire J1939 packet, which includes unnecessary sensor readings, to a SLN, the VN can decode the packet locally and process only the desired sensor readings. It can further reduce the data sent to the SLN by modeling the desired sensor streams using a histogram or any other appropriate modeling technique. As such, when data is

sent to the SLN for fleet-wide analytics, only aggregations (the sensor models) of a subset of sensors (only the desired sensor streams) are sent over the network.

An example of on-board analytics follows: suppose the battery voltage of a vehicle is the parameter of interest, and each individual vehicles' battery voltage is being monitored to make sure they do not exceed a 15V threshold. In this case only the J1939 packets with battery sensor readings need to be decoded, and from these packets only the battery voltage sensor readings need to be considered. The decoded readings are then analyzed for threshold-exceeding values. When a threshold value is exceeded, an event packet can simply be sent to the SLN, indicating a vehicle's battery voltage threshold has been exceeded.

3.4.4 Network Connection

Since the VN is a mobile agent, it must have a network connection via wireless technologies such as Wi-Fi or LTE. Since MQTT (see section 2.7.4.1) is used as the M2M protocol of choice for communication between the perception layer and the middleware layer, the Root Node must give VNs the address of their assigned MQTT broker before they get deployed into the fleet system.

3.5 Server Leader Node

The server leader node (SLN) is found in the middleware layer (see section 3.2). It can be found in the fog or the cloud (which is illustrated in figure 3.1), so it can range from a small resource-constrained device to a cloud cluster with virtually unlimited cloud resources. The SLN is responsible for managing VNs of a geographical region. The geographical region can be of various sizes; for example, all the vehicles in a garage, all airplanes at an airport, or all cargo trucks in a country. The SLN hosts the *Fog-to-Fog* IoT architecture interface (see section 2.1.3.1 for more details) by hosting a MQTT broker for its sub-fleet. This enables the SLN and its VNs to communicate with each other over MQTT. Additionally, the SLN stores all the necessary information of its fleet in its distributed database fragment. Figure 3.3 provides a SysML diagram of the software components involved with the SLN.

3.5.1 Security

I decided to host the MQTT broker on the SLN for security reasons, since hosting the MQTT broker on the SLN makes securing the broker straightforward. Securing the SLN by definition secures the MQTT broker to an extent, instead of needing to deploy an additional middleware layer node to host the MQTT broker, which would require further security configuration.

The SLN is also a good candidate for deploying security mechanisms to meet the data multi-tenancy requirement. For example, suppose a fleet company has a fleet of vehicles that are spread out between two garages. One tenant in this case could be the fleet manager

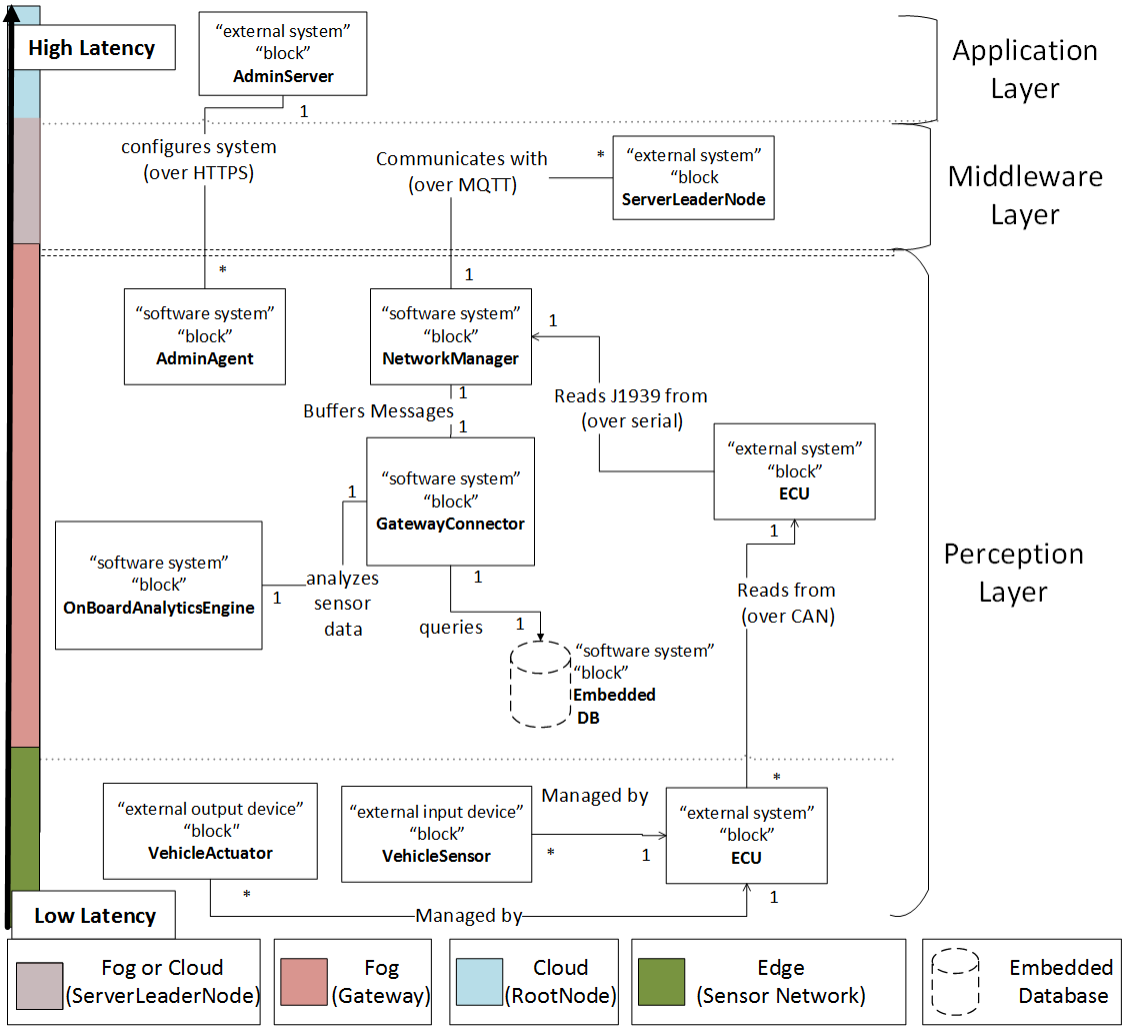

Figure 3.2: SysML [38] diagram of vehicle node. ECU = Electronic Control Unit.

of garage A, and the other tenant is the fleet manager for garage B. Both fleet managers are only allowed to access the vehicles from their assigned garage. To meet this requirement, either a) two SLNs can be deployed to each represent a garage and each fleet manager can only access the vehicles of their assigned SLN or b) a single SLN is deployed and its vehicles are partitioned between both garages by configuring access control mechanisms.

3.5.2 Data Acquisition

Collectively the SLNs of the system host the system's distributed database (DDB) used to store the fleet's vehicle data and other information, by having each SLN host a fragment of the system's DDB. By design, since each SLN manages their own region of vehicles, it makes sense that their DDB fragment only contains vehicle data of their respective regions. This provides data locality and therefore increases availability [92][52]. For example, a North American system could have two SLNs, one for Canada and the other for the United States.

Each SLN would only store vehicle data from the country they were deployed to.

3.5.3 Data Analytics

The SLN is the best candidate for performing fleet-wide data analytics, since a) the SLN has access to all the data of the vehicles of its region (its sub-fleet), b) the SLN generally has more computational resources compared to VNs, which is required, since the number of computations required to analyze a sub-fleet's data is increased by a factor equal to the number of vehicles in the sub-fleet, and c) the SLNs are also able to perform deeper analytics compared to VN, since they are closer to the application layer and therefore have an easier time accessing external data sources that could be used to compliment the fleet-wide analytics. For example, [82] use a vehicle service record database (VSRDB) to perform fault diagnosis.

An example of fleet-wide analytics follows, which is similar to the example in the vehicle node data analytics section 3.4.3: suppose, once again, the battery voltage of a vehicle is the parameter of interest, but this time the time-series nature of each individual vehicles' battery voltage are being monitored and compared to other vehicles in the fleet. The same steps are applied, however, in this case, instead of monitoring thresholds, the VNs model the battery voltage data streams using models that best capture the time-series nature of the data stream, and instead of sending events to the SLN, the models are sent to the SLN. The SLN then performs fleet-wide analysis on each of the models received from its VNs. This example illustrates the power of combining on-board analytics with fleet-wide analytics.

3.5.4 Network Connection

Since the SLN has more computational and networking resources than the VNs, and since MQTT is heavyweight on the server side, I decided that the MQTT broker be hosted as a service on the SLN. Note that the MQTT broker does not need to be physically deployed on the SLN; if required, it could be deployed on an additional middleware node. For example, it may be required to physically separate the SLN's MQTT broker and database fragment, which means both software systems cannot reside on the same physical machine. The MQTT broker can simply be migrated to another middleware layer node to accomplish this. The only configuration required would be to update the reference to the MQTT broker's URL for all fleet nodes referencing it.

In addition to MQTT, the SLN can use more heavy-duty application layer protocols (such as HTTP) to communicate with the rest of the fleet, since the SLNs are not necessarily restricted by the same resource constraints as VNs.

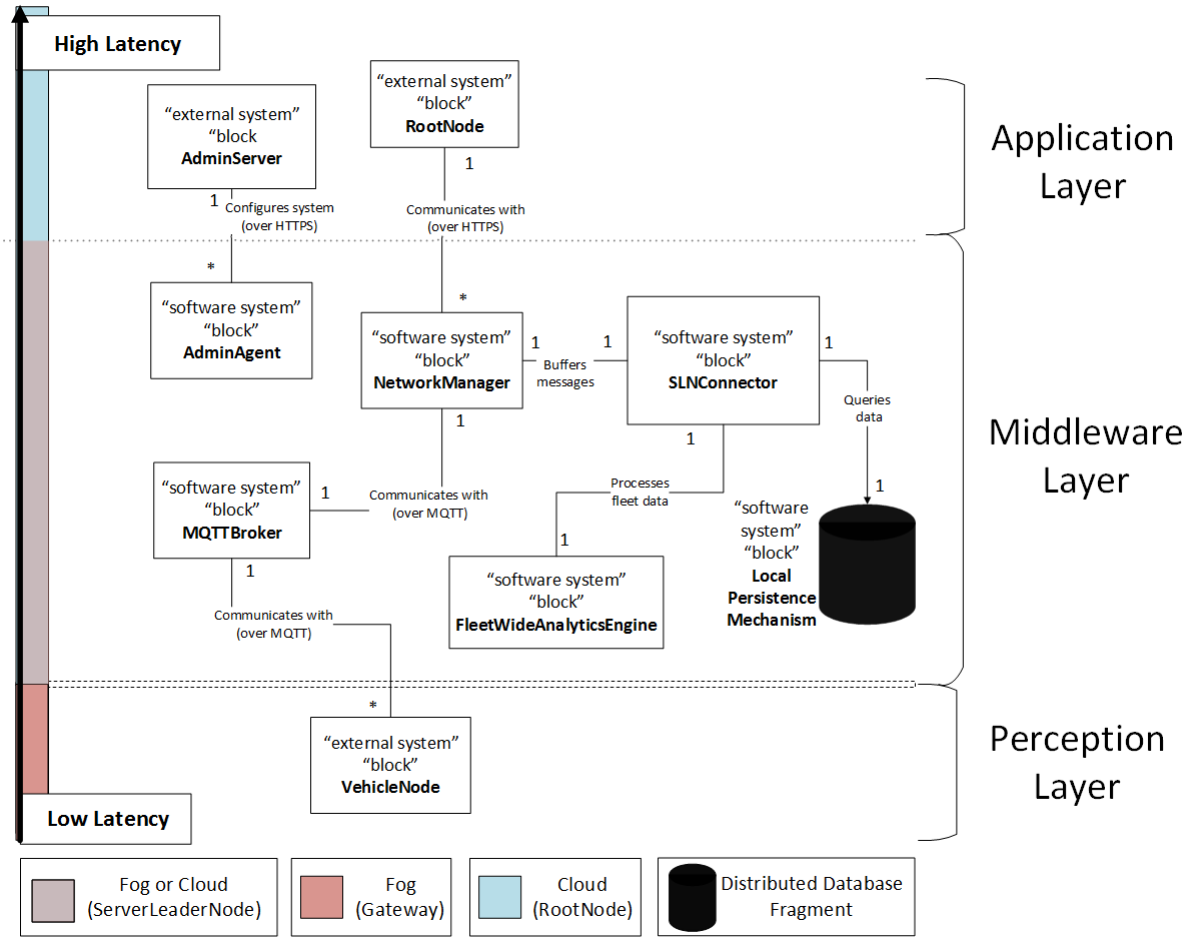

Figure 3.3: SysML [38] diagram of server leader node.

3.6 Root Node

Found in the application layer (see section 3.2), the root node (RN) is responsible for managing the entire fleet system. It is the fleet system's central access point; that is, it provides the interface for IoT applications, connects fleet nodes to the system, and enables fleet-wide administration. In other words, it hosts the *Fog-to-Cloud* IoT architecture interface (see section 2.1.3.1 for more details). Newly deployed fleet nodes request access to the fleet system via the RN's MQTT broker service. All the information required to connect and authenticate fleets nodes can be found in the RN's fleet configuration database. In this database the RN stores all the SLN and VN configuration, deployment, and authentication information, as well as application data, to enable IoT applications. For example, application data could be events published by the SLNs, which are used by fleet management applications to provide visual representations of the status of the fleet. The RN has an administration server and administration agent installed, which allows administrators to remotely configure the software of any fleet node in the system. Since the RN is a cloud node, any generic method for providing internet access to a cloud server is sufficient, unlike the VN (and possibly the SLN) that have specific requirements for network connections.

Crash recovery is outside the scope of this thesis, but it should be considered when deploying the architecture to a live environment, since if the RN fails, all access to the system is lost. Figure 3.4 provides a SysML diagram of the software components involved with the RN.

3.6.1 Security

Since the RN is the central access point to the entire fleet system, it is a good place to enforce fleet system security mechanisms, such as fleet node authentication, SLN region access control, and other access policies. The security of the fleet nodes depends on the security of the RN, which makes securing this node mandatory, since if the RN becomes compromised, the entire system becomes compromised.

3.6.1.1 Fleet Administration

The RN hosts the administration server, which allows fleet administrators to configure the fleet nodes remotely. The RN itself also has an administration agent installed, which allows the administrators to configure the RN via the administration server. It is not sufficient to only be able to configure the RN via the administration server, since if the administration server fails, the entire system would be inaccessible, including the RN. Therefore, an additional access point is required; for example, an SSH server, which would give an administrator full control over the RN when needed. The design and implementation details of the additional access point are outside the scope of this thesis.

3.6.2 Fleet System Interface

The RN hosts all the interfaces of the fleet system, such that any client that requests interaction with the fleet system must first communicate with the RN. Application clients that wish to interact with the fleet system's application servers must first interact with the RN's authentication services. Newly deployed SLNs and VNS requesting access to the fleet system must also first interact with the RN, via the RN's MQTT broker (or any another type server if necessary), before connecting to the fleet. Once authenticated, when new SLNs are deployed, the RN will update its configuration database to allow the SLN to connect to the system. When VNs connect to the system, they will be provided with the list of online SLNs, which are MQTT broker URLs. The VNs may then choose the SLN based on the fastest connection or apply another selection strategy if necessary. Once the VN has been assigned to a SLN, the SLN and RN will update their configuration databases.

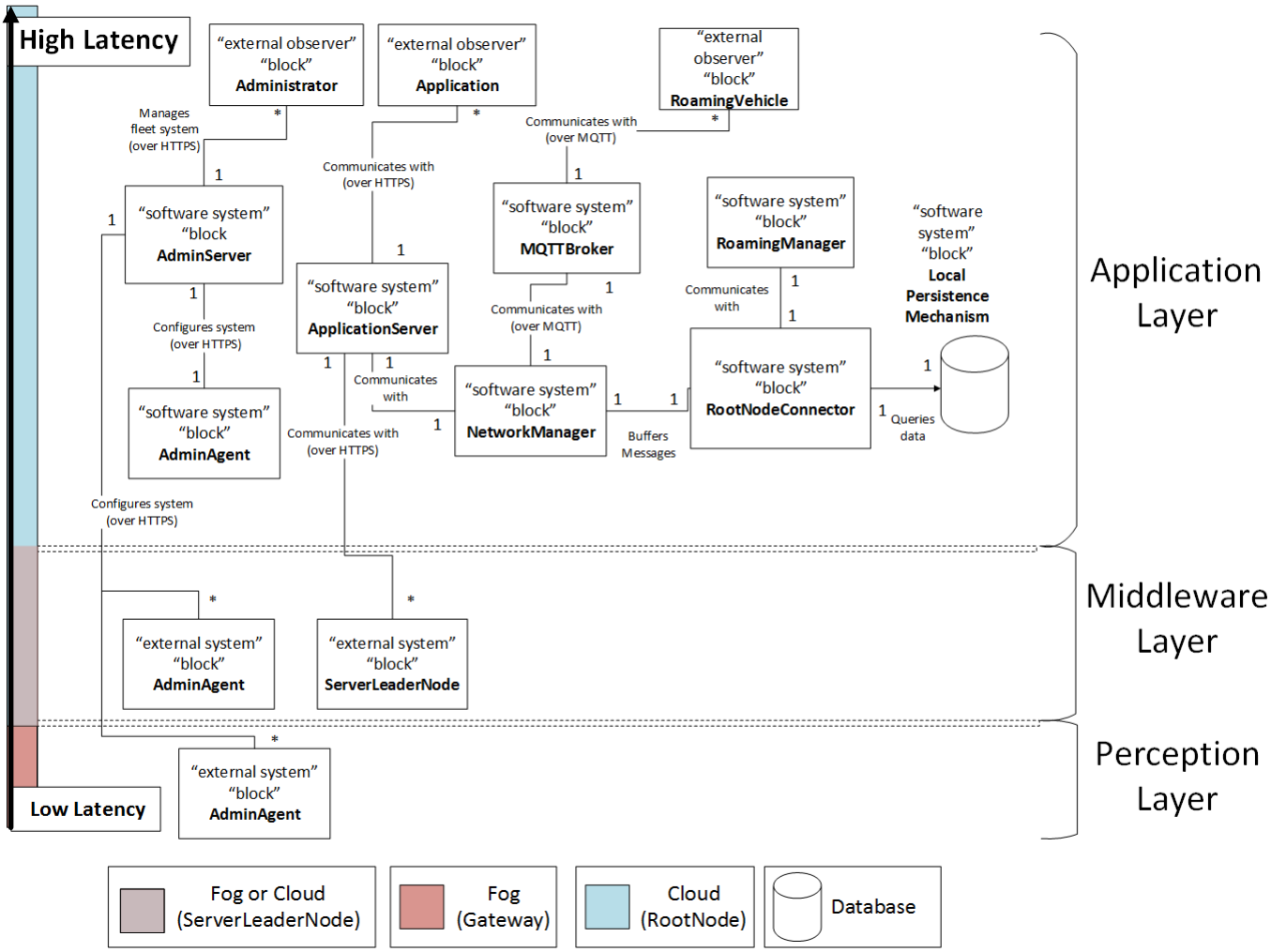

Figure 3.4: SysML [38] diagram of root node.

Chapter 4

Fleet-wide Predictive Maintenance Using Knowledge-base

The present work attempts to improve the sensor selection performed in COSMO [82] by using a fleet-wide unsupervised dynamic sensor selection approach. The algorithm proposed in this work is named Improved Consensus Self-organising Models (ICOSMO).

4.1 Motivation

In regard to the drawbacks of the COSMO approach (which is discussed in section 2.15.4.8), [82] do not provide any evidence to support the absence of no-fault-found (NFF) occurrences. Assuming NFFs occurred, ICOSMO tries to reduce the occurrence rate of NFFs. The following are the possible reasons that COSMO may have failed to detect the faults: a) COSMO's unsupervised sensor selection failed to choose the appropriate sensors needed to detect the undetected fault; b) the best sensors may not be installed on the bus (or are unavailable to the J1939 data acquisition software); and/or c) the distributions of the sensor data streams may have evolved over time (which is mentioned in [36]) for most vehicles, and as a result the original sensor selection is rendered out-of-date. That is, the initially chosen sensors are no longer the best choice for anomaly detection.

Furthermore, the COSMO approach only selects sensors once, so periodically reorganizing the selected sensors should intuitively improved the accuracy of the deviation detection. A few abbreviations, definitions, assumptions, and mathematical notations are required to discuss the design of ICOSMO.

4.2 Abbreviations

Below are a few necessary abbreviations:

- **BBIRA** black box information retrieval algorithm

- **JSD** SAE's J1939 specification document
- **VSRDB** vehicle service record database
- **COSMO** the approach proposed by [82]
- **NFF** no fault found
- **PGN** parameter group number
- **SPN** suspect parameter number
- **SPC** sensor potential contribution
- **SC** sensor contribution
- **ML** machine learning

4.3 Definitions

Below are a few necessary definitions:

Fleet: a group of vehicles that each share the same vehicle model. This definition may be used interchangeably with buses or vehicles.

Sensor Class: a J1939 sensor definition (type of sensor), defined as a PGN-SPN (parameter-group-number, suspect-parameter-number) pair. For example, the sensor class (65262, 110) is for *Engine Coolant Temperature* sensors.

Sensor Instance: a J1939 sensor physically installed on a vehicle. For example, an *Engine Coolant Temperature* sensor found on a vehicle would be a sensor instance of the class (65262,110). In addition to the PGN and SPN attributes, sensor instances have two additional attributes, namely SC and SPC.

COSMO Sensor: a sensor class that has been chosen as a selected sensor in the COSMO approach's deviation detection. As a result, all instances of a COSMO sensor are selected.

Non-COSMO Sensor: a sensor class that has not been chosen as a selected sensor in the COSMO approach's deviation detection. As a result, all instances of a COSMO sensor are not selected.

Deviation: a sensor instance's z-score has exceeded a user-defined deviation threshold, indicating the sensor instance is deviating from the rest of the fleet.

SC: a COMSO sensor instance's estimated ability to benefit the deviation detection when included in the COSMO model. A sensor class that contributes more than another will enable deviations to be detected with more accuracy (reducing the false positive and false negative (or NFF) rates).

SPC: a non-COSMO sensor instance's estimated potential ability to benefit deviation detection if it were to be included in the COSMO model. A sensor instance that has high potential contribution is a sensor found/installed on the vehicle and that has been estimated to be part of a component that is frequently repaired during a period when no deviation was detected.

Candidate Sensors: non-COSMO sensors that most of their sensor instances have high potential contribution

Stale Sensors: COSMO sensors where most of their sensor instances have low contribution

Fault-involved Sensor: a sensor class that is directly or indirectly linked to a fault on a vehicle. That is, this type of sensor will have a noticeable change in its output behavior when a fault occurs. For example, if a component used to heat a vehicle's internal cab breakdowns, the internal thermostat would be a fault-involved sensor, since it would output cooler interior temperatures indirectly as a result of the fault.

BBIRA: ICOSMO requires a specific information retrieval algorithm. This algorithm's role is to find/estimate sensor instances involved in a repair (fault-involved sensors). With a repair record as a query and the J1939 specification as the index, this algorithm estimates fault-involved sensors.

4.4 Assumptions

ICOSMO makes the following assumptions:

Assumtpion 1. A VSRDB, with repair records that describe details about faults that were repaired, is accessible (this is not unreasonable, since [82] had access to such a database).

Assumption 2. The repair records in the VSRDB are associated with true faults (that is, the records detailing the fault and the steps taken to repair them are linked to actual failures, not preventive repairs but instead reactive repairs).

Assumption 3. An information retrieval algorithm exists, which takes a mechanic's repair record as a query, searches its indexed J1939 specification document, and estimates the sensor classes involved in the failing/faulty components that the repair fixed (fault-involved sensors). Note that a BBIRA is used in this work to simulate this assumption.

Assumption 4. All buses in a fleet are of the same model, and each bus shares similar daily travel routes.

Assumption 5. Sensors physically located on (or involved with) a component have more chances of being successful at detecting a deviation in response to a component's failure, compared to sensors physically far away from (or not involved with) the component. For Example, sensors found on aN engine have more chances of detecting an engine fault compared to a backdoor sensor.

Assumption 6. Vehicles only implement one sensor instance of a sensor class. That is, they do not have redundant sensor instrumentation (it is beyond the scope of this work).

4.5 Sensor Set Definitions

Some definitions are required to explain ICOSMO's methodology. All the sensors defined in the JSD (see section 2.9.3 for more information on J1939) are placed into sensor sets. The elements of the sets, sensor instances or sensor classes, are represented identically as PGN-SPN pairs; that is, 2-dimensional points of the form $element = (x,y)$.

- N is number of buses in the fleet of a SLN, which is equivalent to $|B|$
- M is number of unique sensor classes found on the buses, which is equivalent to $|I|$
- B is an MxN matrix of sensor instances in the fleet, where the N columns represent vehicles, the M rows represent sensor classes, and each cell is a sensor instance. The row i is the set of sensor instances sharing the ith sensor class, or (without loss of generality) sharing the sensor class $i = (x,y)$; that is, not necessarily the row with index i; where each sensor instance, a matrix cell, are found on a different vehicle. The column j is the set of sensor instances sharing the jth vehicle, or without loss of generality, sharing the vehicle with id j, i.e. not necessarily the column with index j, where each sensor instance, a matrix cell, has a different sensor class.

$$B = \begin{bmatrix} B_{11} & B_{12} & B_{13} & \dots & B_{1N} \\ B_{21} & B_{22} & B_{23} & \dots & B_{2N} \\ \dots & \dots & \dots & \dots & \dots \\ B_{M1} & B_{M2} & B_{M3} & \dots & B_{MN} \end{bmatrix}$$

- B_i is vehicle i's sensor instances, $B_i = \begin{bmatrix} B_{1i} \\ B_{2i} \\ \dots \\ B_{Mi} \end{bmatrix}$
- $\hat{B}_i$ is ith sensor class' sensor instances, $\hat{B}_i = \begin{bmatrix} B_{i1} & B_{i2} & \dots & B_{iN} \end{bmatrix}$
- $B_{ij} = \hat{B}_{ji}$, where both refer to sensor instance of the jth sensor class on vehicle i
- J is set of unique sensors classes defined in JSD
- U is set of unique sensors classes not installed, or unavailable, for each vehicle
- I is set of unique sensors classes installed on each vehicle
- E is set of unique non-COSMO sensors
- C is set of unique COMSO sensors
- O is hidden set of optimal COSMO sensors classes (set that maximizes deviation detection performance)

4.5.1 Sensor Set Comparison

Since the sensor instances and classes are represented identically as PGN-SPN pairs in the sets defined in section 4.5 of this chapter, a definition for the approach taken to compare elements of these sets (sensor classes and sensor instances) is required. When performing sensor class comparisons, two sensor classes i and j with the same PGN (x) and SPN (y) are equivalent. That is, if $i = (x,y)$ and $j = (x,y)$, then, $i = j$.

Since the set B represents a matrix of sensor instances, but its elements are PGN-SPN pairs, which can be interpreted as sensor classes, using the index notation on the set B will differentiate sensor instance notation from sensor class notation. For example, for some $k = (x, y)$ and $B_{ik} = (x, y)$, k and (x, y) denote a sensor class with PGN $= x$ and SPN $= y$, while B_{ik} denotes a sensor instance of the class k. That is, B_{ik} has PGN $= x$, SPN $= y$, and has a SC and SPC attribute as well. Note that mathematically $k = (x, y) = B_{ik}$, but B_{ik} also refers to a sensor instance object, not only referring to an integer pair.

4.5.2 Sensor Set Properties

All the sensors installed on the vehicle and those that are not installed on the vehicle are assumed to be defined in the JSD. Naturally the sensors analyzed by the COSMO method must also be defined in the JSD. Otherwise, without a defined standard to follow for reading and decoding sensor data, it would not be possible. The non-COSMO sensors can either be installed on the bus or not be present. All of this is expressed by the expressions in 4.1 below and illustrated in figure 4.1:

$$\begin{aligned} U \cup I &\equiv J \\ E &\subseteq I \\ E &\subseteq U \\ C \cup E &\equiv J \\ C &\subseteq I \end{aligned} \tag{4.1}$$

All the COSMO sensors must be found on the vehicle, since sensors that are not installed on the vehicle cannot be included in the set of selected sensors. Furthermore, a sensor cannot be included in the COSMO model and excluded at the same time. All of this is expressed by the expressions in 4.2 below:

$$\begin{aligned} C \cap U &\equiv \emptyset \\ E \cap C &\equiv \emptyset \\ \overline{C} &\equiv E \end{aligned} \tag{4.2}$$

Each bus in the fleet shares the same model, and therefore each bus has the same type of sensors installed on-board, which can be expressed by the expressions in 4.3 below:

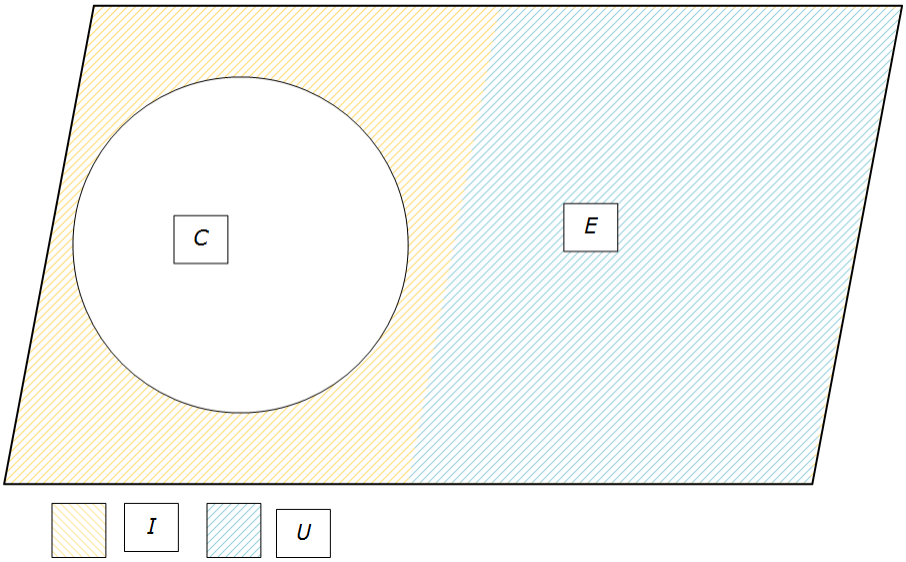

Figure 4.1: Sensor set ven diagram, where the set I is the yellow area, U is the blue area, C is the white circle, and E is everything outside the white circle.

$$\begin{aligned} &\forall i \in \{1, 2, ..., N\} \\ &B_i \cap U \equiv \emptyset \\ &B_i \equiv I \end{aligned} \tag{4.3}$$

4.6 Methodology

ICOSMO is designed in a data-driven fashion for conducting predictive maintenance (see section 2.10.1.3 for more information), since mostly sensor data and historical data are accessible. Using BBIRA, ICOSMO dynamically adjusts COSMO sensors over time, by removing stale sensors from the COSMO sensors and adding candidate sensors to the COSMO sensors, which is done by introducing the following metrics for each sensor instance: sensor contribution (SC) and sensor potential contribution (SPC). ICOSMO is separated into three steps:

1. Find the sensor classes involved with a faulty component using the BBIRA
2. Modify the SC and SPC of sensor instances based on their estimated ability (or potential ability) of finding deviations
3. Periodically re-organize the COSMO sensors by removing/adding sensor classes from/to the COSMO sensors by analyzing the average SC and SPC of a sensor class' sensor instances:
    - When the majority of a COSMO sensor's instances are stale, remove them from the selected sensors.

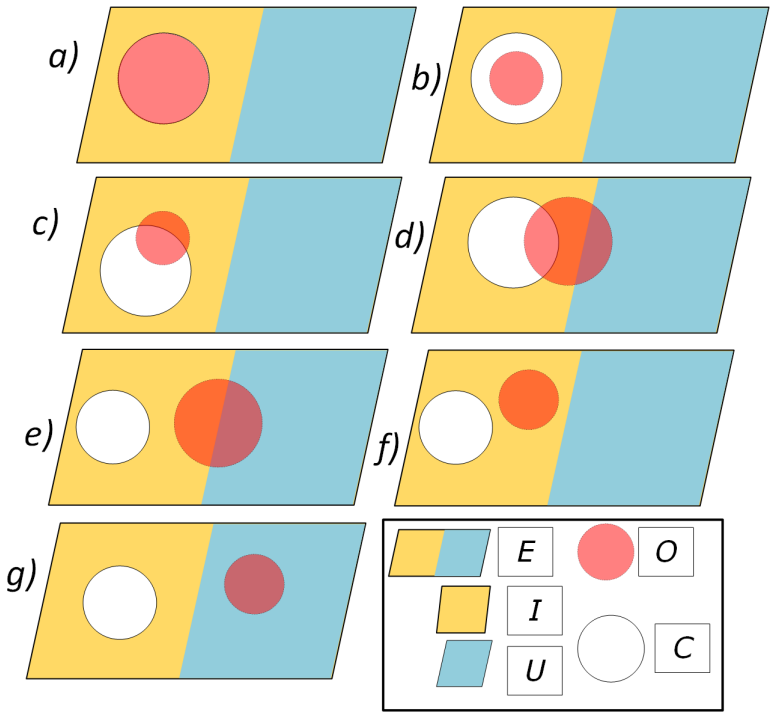

Figure 4.2: All cases of possible optimal set O's configurations.

- When the majority of a non-COSMO sensor's instances are candidates, add them to the selected sensors.

4.7 Improving Deviation Detection

ICOSMO attempts to select the best COSMO sensors. In theory it should choose the optimal set of sensors O, which can shift over time, due to the nature of sensor degradation, change in seasons, or other external environmental factors. Visually, O can be represented as the red circle in figure 4.2. ICOSMO will always be trying to achieve $C \equiv O$ if possible, $C \subset O$ when there is a lack of available optimal sensors, or the next optimal sensor set O° which is available on the vehicle when O is completely unavailable (illustrated in diagram g in figure 4.2). ICOSMO's goal can be represented as follows in the expression 4.4.

$$\text{ICOSMO Goal} = \begin{cases} C \equiv O, & \text{if } O \cap U \equiv \emptyset \\ C \subset O, & \text{if } O \cap U \neq \emptyset \text{ and } O \cap I \neq \emptyset \\ C \equiv O^{\circ}, & \text{if } O \cap I \equiv \emptyset \end{cases} \tag{4.4}$$

ICOSMO tries to satisfy the expression 4.4 by doing the following:

- adding candidate sensors to the selected features (move from E to C)
- removing stale sensors from the selected features (move from C to E)

4.7.1 Optimal Sensor Set Analysis

Satisfying the expression 4.4 is not trivial given all possible sensor data distributions and selected feature combinations (see section 4.7.1.1 for more details on complexity analysis). For example, an expensive (but optimal) sensor that is not installed on a vehicle for financial reasons will prevent the expression $C \equiv O$ from being satisfied.

Referring to figure 4.2, for diagram a, it shows the case when $C \equiv O$, which is ICOSMO's goal. ICOSMO is to do nothing in this case. Diagram b illustrates the case when $O \subset C$. In this case, ICOSMO should remove stale sensors from C, those in $\overline{O} \cap C$, until $C \equiv O$. In all other cases (other than a and b) ICOSMO will aim to achieve $C \equiv O$ if possible; that is, when all optimal sensors are available on the bus ($O \cap U \equiv \emptyset$). Otherwise ICOSMO will aim to achieve $C \subset O$. ICOSMO will do so by removing stale sensors from C (and adding them to E) and adding candidate sensors to C (removing them from E). Diagram c illustrates the case that O is partially contained in C, and that the O is entirely composed of sensor classes found on-board (or available on) the vehicle. The candidate sensors that ICOSMO should consider adding to C are those in $O \cap E$. Diagram d illustrates the case when some of the optimal sensors are not available (either proprietary encoding or not installed on the vehicle, for example). When the unavailable optimal sensors ($O \cap U$) become candidate sensors, it should be suggested to fleet managers to install these sensors onto buses. Diagram e illustrates the case where no optimal sensors are selected. Diagram f illustrates the case where no optimal sensors are selected, but they are all available on the bus. Diagram g illustrates the case where there are not any optimal sensors available. Although the optimal sensor selection is impossible without installing a new sensor, ICOSMO will attempt to achieve the next best sensor selection set.

4.7.1.1 Computational Complexity

For complexity analysis purposes, sensor selection can be effectively treated as feature selection, since the data streams of sensors can be represented as features of a ML model. According to [41], adding multiple individually redundant features to a ML model can improve the ML model's classification by reducing noise and increasing class separation. In other words, it is necessary to consider feature sets when evaluating the prediction power of a ML model. Therefore, it is important to search for the best subset of features for a ML model. Unfortunately, this becomes infeasible as the number of features increase, since there are an exponential number of possible feature subsets [100]. That is, searching for the optimal feature subset is NP-hard.

4.7.2 Finding Sensors Involved in a Fault

ICOSMO attempts to find fault-involved sensors when a repair occurs by using a BBIRA. The goal of the sensor retrieval is to identify all the fault-involved sensors with respect to a component that had a failure. The logic is as follows: sensors in close proximity to a component that failed, either physically installed on the component or closely involved

with the component, should have better deviation detection accuracy as a COSMO sensor compared to sensors that are far away from (or not involved with) the failure. For example, sensors installed on an engine intuitively should more accurately contribute to deviation detection when an engine begins to exhibit faulty behavior, compared to sensors installed on a door.

When a repair record, r, is inserted into the VSRDB, ICOSMO uses the mechanic's comments and the date (found in r) as a query to the BBIRA. The BBIRA then returns a set of estimated fault-involved sensors, X, which is an estimation of the sensors involved with the component that was repaired by r. The scope of the sensors found in X is limited by the sensors found in J, since the BBIRA by definition searches an indexed JSD. This can be expressed using the expression 4.5 below, where n is the number of desired sensors to fetch .

$$\begin{aligned} &X \Leftarrow \text{sensorRetrievalAlgorithm}(r, n) \\ &|X| = n \\ &\forall s \in X, s = (pgn, spn) \\ &X \subseteq J \end{aligned} \tag{4.5}$$

For more details on information retrieval, see section 2.14. The sensor retrieval algorithm is an information retrieval algorithm that parses the JSD into an index. It uses its index to find all the sensor classes involved in a fault on some vehicle, such that their descriptions most closely match the information found in the query. The query is a repair record, r, which was created by a mechanic in response to fixing a fault, that describes all the details of fault that was repaired. It is beyond the scope of this work to define the implementation of such an algorithm. Since ICOSMO requires such a BBIRA, I will therefore run my ICOSMO experiments by simulating the BBIRA. The goal of this work is to investigate if further research into designing a JSD-based fault-involved sensor information retrieval algorithm is worth looking into.

4.7.2.1 Simulating the Fault-involved Sensor Retrieval Algorithm

Since a simulation should abstract the actual implementation but output the desired behavior, the simulated BBIRA should estimate the fault-involved sensors the same way as an actual implementation. To accomplish this, additional parameters are used for the BBIRA. The simulated BBIRA will take a desired recall or desired precision parameter. Only one of the parameters is used, since both cannot be satisfied. That is, this simulation will focus on satisfying either precision or recall. This design choice is made since an information retrieval algorithm will estimate objects of a desired class (in this case fault-involved sensors), so providing desired recall or precision will simulate the same behavior as a BBIRA with the given precision or recall. The definitions of recall and precision are presented in equation 4.6:

$$precision = \frac{TP}{TP + FP}, \quad recall = \frac{TP}{TP + FN} \tag{4.6}$$

Since the desired precision or recall are provided, it means the simulated BBIRA must return an estimated set of fault-involved sensors that satisfy the expression 4.6 above. This leads to the next abstraction of the BBIRA. It would normally expect a repair record as a query, but since the number of TP, FP, and FN estimations are required, it is therefore necessary to have the following inputs and parameters:

Input

- P is set of fault-involved sensors with respect to the component failure that was repaired and described in the VSRDB by a repair record r entered by a mechanic.
- $\overline{P}$ is set of non-fault-involved sensors with respect to the component failure described by r.

Note that $J \equiv P \cup \overline{P}$.

Parameters

- n is an integer, the desired number of fault-involved sensors to be estimated.
- *recall* is a real number from 0 to 1 that represents the desired recall of the estimation result set.
- *precision* is a real number from 0 to 1 that represents the desired precision of the estimation result set.

Label Estimation

A definition of what a TP, FP, and FN prediction is in this context is required. Let X_i denote the ith estimated fault-involved sensor. Estimated fault-involved sensors in X will have two binary labels, ν and ω, where $\nu(X_i)$ is the actual classification and $\omega(X_i)$ is the estimated classification of the ith sensor. Let "+" and "-" denote a fault-involved sensor and non-fault-involved sensor, respectively, with respect to the repair record r. That is,

$$\forall p \in P, \nu(p) = \text{“+”}$$
$$\forall p \in \overline{P}, \nu(p) = \text{“-”}$$

Table 4.1 illustrates how to determine whether a sensor label estimation x is a TP, FP, FN, or TN prediction:

$\omega(x)/\nu(x)$	ν: "+"	ν: "-"
ω: "+"	TP	FP
ω: "-"	FN	TN

Table 4.1: Information retrieval algorithm possible predictions.

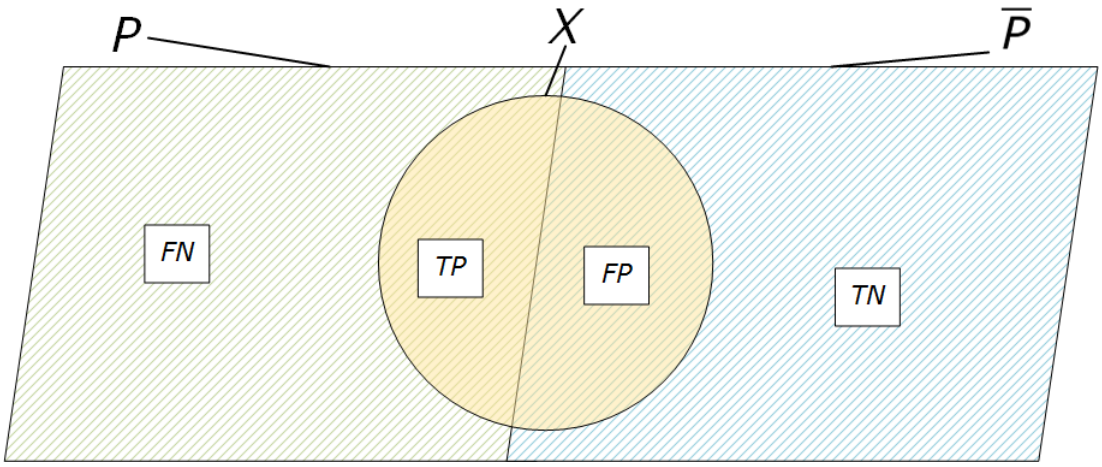

Figure 4.3: Where P are all the fault-involved sensors, $\overline{P}$ are all the non-fault-involved sensors, and X are the sensors estimated by the BBIRA.

4.7.2.2 Methodology

To compute X to simulate the BBIRA, the TP, FN, and FP predictions must be expressed with respect to the input, which is done as follows by referring to figure 4.3:

$$S \leftarrow P \cap X \equiv TP$$
$$P \cap \overline{X} \equiv FN$$
$$\overline{P} \cap X \equiv FP$$

The precision and recall expressions (see equation 4.6) can now be expanded and expressed with respect to P, $\overline{P}$, X, and S:

$$precision = \frac{TP}{TP+FP} = \frac{|S|}{|S \cup (\overline{P} \cap X)|} = \frac{|S|}{|(P \cap X) \cup (\overline{P} \cap X)|} = \frac{|S|}{|\mathrm{X}|}$$
$$recall = \frac{TP}{TP+FN} = \frac{|S|}{|S \cup (P \cap \overline{X})|} = \frac{|S|}{|(P \cap X) \cup (P \cap \overline{X})|} = \frac{|S|}{|\mathrm{P}|}$$

The expression 4.7 must be satisfied to meet the desired recall or precision.

$$\begin{aligned} precision &= \frac{|S|}{|X|} \\ recall &= \frac{|S|}{|P|} \end{aligned} \tag{4.7}$$

To simulate X, computing the number of sensors from P, denoted by i, and from $\overline{P}$, denoted by j, is required. With a bit of algebra, i and j can be expressed as follows in equations 4.8 and 4.9 below for the BBIRA with desired precision and desired recall, respectively:

$$\begin{aligned} i &= precision * n = |S| \\ j &= n - i \end{aligned} \tag{4.8}$$

$$\begin{aligned} i &= recall * |P| = |S| \\ j &= n - i \end{aligned} \tag{4.9}$$

To simulate the non-deterministic nature of an information retrieval algorithm, a random number of samples from the set P and $\overline{P}$ are chosen, which respect equations 4.8 and 4.9, for building the estimated set X. That is, i random samples are chosen from P to be added to the result set X, and j random samples are chosen from $\overline{P}$ to be added to the result set X. The pseudo code for this procedure is shown in algorithm 1.

Algorithm 1 Simulating fault-involved sensor information retrieval algorithm.

```
1:  procedure IRA_RECALL(P, P̄, recall, n)
2:      i = recall * |P|
3:      j = n - i
4:      X = IRA_helper(P, P̄, i, j)
5:      return X
6:  procedure IRA_PRECISION(P, P̄, precision, n)
7:      i = precision * n
8:      j = n - i
9:      X = IRA_helper(P, P̄, i, j)
10:     return X
11: procedure IRA_HELPER(P, P̄, i, j)
12:     X = {}
13:     P = random permutation of P
14:     P̄ = random permutation of P̄
15:     for k = 0 to i do
16:         X = X ∪ P_k
17:     for k = 0 to j do
18:         X = X ∪ P̄_k
19:     return X
```

4.7.3 Adjusting Sensor Contribution and Potential Contribution

For notation purposes, let "sensors involved with a repair r" denote the set of sensor classes found on the repaired vehicle that were involved with a fault that has been repaired by

the repair described by r. When a repair record, r, is inserted into the VSRDB, it gives ICOSMO the chance to adjust the SC and SPC of the sensors involved with it. Let $i = r_{vid}$ be the vehicle's id. Let Y denote all the COSMO sensors found on bus i and related to r; that is, $Y = B_i \cap X \cap C$. Let Z denote all the non-COSMO sensors found on bus i related to r; that is, $Z = B_i \cap X \cap E$. For notation purposes, let "sensor instances in Y" denote the set of sensor instances for each sensor class in Y that are found on vehicle i; that is, $\forall B_{ij}$, where $j \in Y$. A similar notation may be used with the set Z.

There are two cases: either the deviation detection has detected the fault or it has not. In ICOSMO, a fault is defined as being detected when at least one sensor instance in Y has deviated in the past (within a short time period). The short period is a user-defined parameter that defines how many days in the past should be considered when searching for deviations. A fault has not been detected if this is not the case.

Note that the pseudo code for adjusting the SC and SPC is found in algorithm 2.

Case 1: Deviation Detected

In this case at least one sensor instance in Y has detected the fault. The expression 4.10 indicates how the SC and SPC of the sensor instances are adjusted, where Y^d denotes each instance in Y that detected the fault, and $\overline{Y^d}$ denotes those that did not:

$$\begin{aligned} &\forall s \in Y^d, increaseSC(B_{is}) \\ &\forall s \in \overline{Y^d}, decreaseSC(B_{is}) \\ &\forall s \in Z, decreaseSPC(B_{is}) \end{aligned} \tag{4.10}$$

Case 2: No Deviation Detected (NFF)

In this case no instances in Y detected the fault. The expression 4.11 indicates how the SC and SPC of the sensor instances are adjusted:

$$\begin{aligned} &\forall s \in Y, decreaseSC(B_{is}) \\ &\forall s \in Z, increaseSPC(B_{is}) \end{aligned} \tag{4.11}$$

4.7.4 Dynamic Sensor Selection

ICOSMO dynamically adjusts the COSMO sensors by estimating the average deviation detection performance (and estimated performance) of each sensor class. It does so by computing the average SC and SPC among the sensor instances for each sensor class and identifying stale sensors, which should be removed from the COSMO sensors, and candidate sensors, which should be added to the COSMO sensors. It is worth noting that ICOSMO only searches for stale and candidate sensors when for some sensor instance, its SPC has been increased or its SC has been decreased. The pseudo code for adjusting the sensor selection is found in algorithm 3.

Algorithm 2 Updating sensor deviation detection ability.

1: **procedure** onRepairInsert(r,X) ▷ Repair record r, X is the estimated fault-involved sensor set for bus $r_{\texttt{vid}}$
2: $i = r_{\texttt{vid}}$ ▷ vehicle id
3: $Y = B_i \cap X \cap C$ ▷ fault-involved cosmo sensor instance of vehicle i
4: $Z = B_i \cap X \cap E$ ▷ fault-involved non-cosmo sensor instance of vehicle i
5: $Y^d = \{\}$ ▷ sensor instances that detected fault
6: $\overline{Y^d} = \{\}$ ▷ NFF sensor instances
7: **for** s `in` Y **do**
8: **if** `deviationExists(`B_{is}`,`r`.date)` **then**
9: $Y^d = Y^d \cup s$
10: **else**
11: $\overline{Y^d} = \overline{Y^d} \cup s$
12: **if** $\mid Y^d \mid > 0$ **then** ▷ detected fault?
13: **for** s `in` Y^d **do**
14: increase contribution of sensor instance B_{is}
15: **for** s `in` $\overline{Y^d}$ **do**
16: decrease contribution of sensor instance B_{is}
17: **for** s `in` Z **do**
18: decrease potential contribution of sensor instance B_{is}
19: **else** ▷ failed to detect fault
20: **for** s `in` Y **do**
21: decrease contribution of sensor instance B_{is}
22: checkStaleSensorRemove(s)
23: **for** s `in` Z **do**
24: increase potential contribution of sensor instance B_{is}
25: checkCandidateSensorAdd(B_{is})

4.7.4.1 Stale Sensors

To find out whether a COSMO sensor, i, is not contributing to the accuracy of the deviation detection (that is, on average its sensor instances are failing to detect faults), ICOSMO analyzes the average SC of $\hat{B}_i$ as follows:

$$\mu = \frac{1}{N} \sum_{j=1}^{N} \sigma(\hat{B}_{ij}) \tag{4.12}$$

where $\sigma(\hat{B}_{ij})$ refers to the SC of sensor instance $\hat{B}_{ij}$. i is performing poorly (that is, identified as a stale sensor), if μ is below a user-defined threshold, in which case i is removed from the COSMO sensors.

4.7.4.2 Candidate Sensors

The same process applied to search for stale sensors (see section 4.7.4.1) is applied to search for candidate sensors, but with a few changes. With respect to equation 4.12, the changes are as follows: a) $\sigma(\hat{B}_{ij})$ refers to the SPC of the sensor instance $\hat{B}_{ij}$ instead of to the SC of the sensor instance and b) i is added to the COSMO sensors when μ is *above* a user-defined threshold instead of when μ is below the threshold.

Algorithm 3 Self-organizing sensor selection.

```
1:  procedure checkCandidateSensorAdd(s)          ▷ s non-COSMO sensor class
2:      θ = average potential contribution threshold
3:      μ = (1/N) Σ_{i=1}^{N} potential contribution of B̂_si     ▷ mean SPC for sensor class s
4:      if μ > θ then      ▷ # threshold met? ie sensor class s a candidate sensor class?
5:          if s ∈ U then                                  ▷ sensor not installed on bus?
6:              sendMessageToFleetManager("/install/sensor/suggestion",s)
7:          else                    ▷ add sensor class s to COSMO's selected features
8:              sendMessageToAllVehicles("/add/sensor/COSMO",s)
9:              E = E - s
10:             C = C ∪ s
11: procedure checkStaleSensorRemove(s)          ▷ s non-cosmo sensor class
12:     θ = average contribution threshold
13:     μ = (1/N) Σ_{i=1}^{N} contribution of B̂_si              ▷ mean SC for sensor class s
14:     if μ < θ then      ▷ # threshold met? ie sensor class s a stale sensor class?
15:         sendMessageToAllVehicles("/remove/sensor/COSMO",s)
16:         E = E ∪ s
17:         C = C - s
```

4.8 Knowledge Discovery

In addition to attempting to increase the accuracy of COSMO's deviation detection, ICOSMO may also offer knowledge discovery, such as faulty sensor detection or sensor

installation suggestions, by data mining the SCP and SC of the sensor instances in the fleet. Note that in both cases, it is outside of the scope of this work to investigate these research avenues.

4.8.1 Faulty Sensor Detection

Since each bus share the same model and the null-hypothesis of the COSMO approach is that all sensor instances of a class have the same output data distribution, then each sensor instance should contribute similarly to deviation detection. Although faults typically are unpredictable and do not occur at the same frequency on vehicles of the same model, it is reasonable to assume that healthy sensor instances of a sensor class will have more similar SC than faulty sensor's SC.

Let i be some COSMO sensor and all sensor instances but one in $\hat{B}_i$ have a similar SC. Let $\hat{B}_{ij}$ be the sensor instance whose SC is deviating from the rest of $\hat{B}_i$. There are two cases:

Case 1 - High SC: $\hat{B}_{ij}$ has more SC than the rest of $\hat{B}_i$. In other words, $\hat{B}_{ij}$ is outperforming the rest of the fleet at fault detection. An explanation for this could be the component hosting sensor i on bus j has been replaced; that is, sensor instance $\hat{B}_{ij}$ has been physically replaced with a new sensor instance. In this case the new sensor $\hat{B}_{ij}$ may be reporting much more accurate readings compared to the rest of $\hat{B}_i$. In this case, no action is required.

Case 2 - Low SC: $\hat{B}_i j$ has much lower SC than the rest of $\hat{B}_i$. In other words, $\hat{B}_{ij}$ is under-performing compared to the rest of its fleet at detecting faults. An explanation for this could be that $\hat{B}_{ij}$ is faulty, such that either $\hat{B}_{ij}$ is reporting inaccurate or garbage readings, or $\hat{B}_{ij}$ is not reporting readings at all. In either case, $\hat{B}_{ij}$ is not helpful at detecting faults compared to the rest of its fleet. In this case, action can be taken by notifying a fleet manager that $\hat{B}_{ij}$ may be faulty.

4.8.2 Suggesting Sensor Installation

It may be the case that an unavailable sensor class is flagged as a candidate sensor, and although it may help the deviation detection accuracy if it were added to the COSMO sensors, it is not possible since the sensor is not accessible by the data acquisition hardware. This suggests that installing the candidate sensor on each vehicle may be financially advisable for the fleet manager. In this case, action can be taken and the fleet manager can be notified that installing the candidate sensor on each vehicle of the fleet would benefit the predictive maintenance system.

Chapter 5

Implementation and Results

5.1 Predictive Maintenance Architecture System Prototype

A minimally viable prototype (MVP) of the architecture mentioned in section 3 has been implemented and is running in a live environment. The MVP does not implement ICOSMO (which is discussed in chapter 4).

The MVP is targeted for creating an IoT predictive maintenance fleet management system for the public transport buses of the Société de Transport de l'Outaouais (STO), Gatineau, Canada. Each bus will have a gateway installed, which reads sensor data and performs lightweight analytics. The goal is to discover novelties and to provide this information to the fleet managers to help them make better maintenance decisions. With each bus equipped with a gateway, fleet-wide data analysis will be possible. This enables the possibility of discovering novelties that would not be obtainable when only monitoring individual vehicles.

5.1.1 STO Requirements Gathering

I had many meetings with the STO to gather requirements for the MVP. I learned that occasionally the voltage of the 24V bus batteries would fluctuate. I, therefore, needed an uninterruptible power supply (UPS). A waterproof casing was also required to avoid water damage. The buses used the SAE's J1939 protocol for heavy-duty machinery (see section 2.9.3). There were no Ethernet connections on the bus, so the MVP needed wireless internet connection.

5.1.2 Gateway Implementation

From the requirements, I created the MVP. I installed the hardware in a waterproof plastic container (which can be seen in figure 5.2) containing the following hardware: a Raspberry

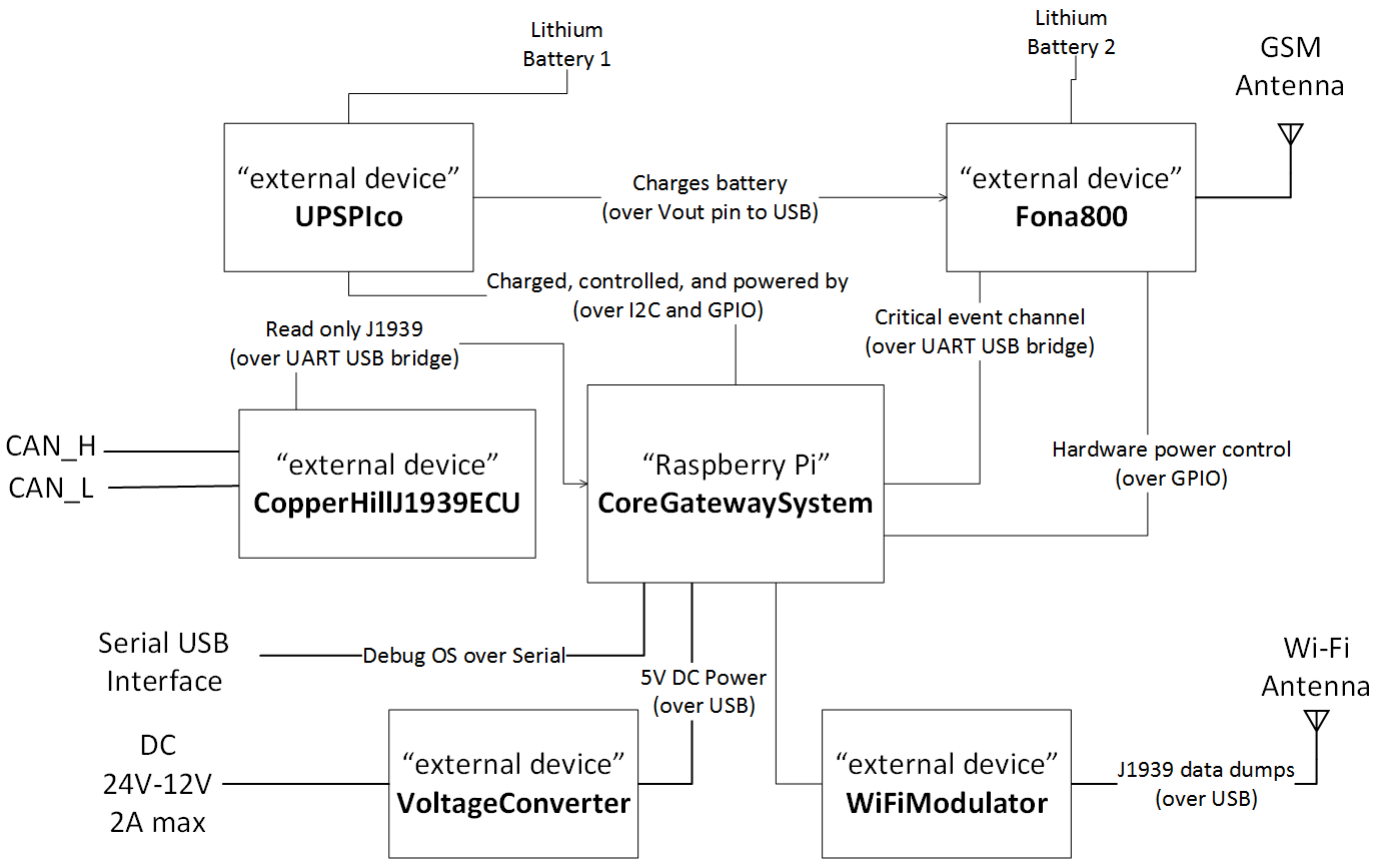

Figure 5.1: Hardware SysML [38]-style diagram of gateway.

Pi as the operating system, a UPS PIco, an Adafruit FONA 800 for a global system for mobile communications (GSM) network connection, a USB Wi-Fi antenna, a 24V-to-5V converter for power, and a Copperhill J1939 electronic control unit (ECU). A complete list of hardware components used to build the gateway is found in table B.1. Figure 5.1 illustrates a SysML-style diagram of the gateway's hardware components. Furthermore, I chose the administrative software Meshcentral, an open source software package for managing a private network of machines [65]. See Appendix B.1 for additional details on the gateway's hardware.

5.1.2.1 Gateway Test Cases

To deploy the prototype into a live environment, test cases were compiled to make sure the gateway would meet the following requirements: have persistent internet connection, always be connected to Meshcentral, possess the ability to read (but not write) from the test vehicle's J1939 network, be resistant to vehicle battery voltage fluctuations, prioritize the STO Wi-Fi connection over GSM when possible.

From a security perspective, the only way one could gain access to the gateway was via Meshcentral. If the gateway's software becomes compromised, the fleet system should

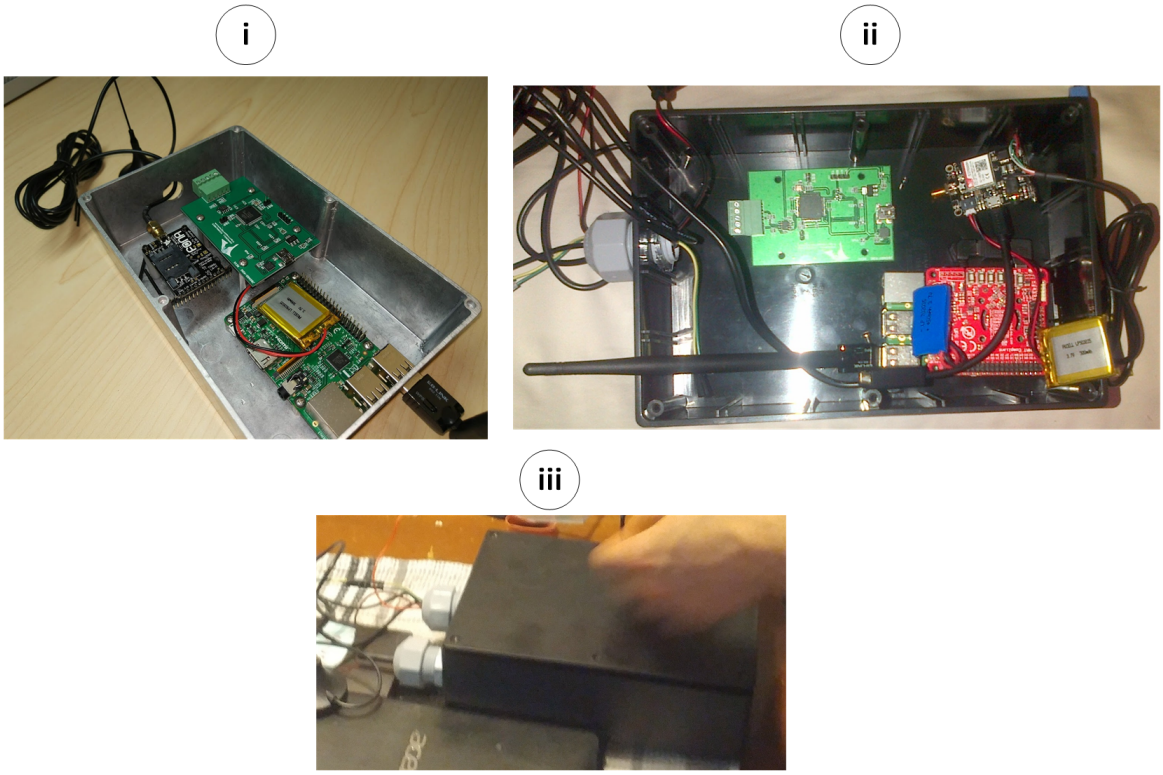

Figure 5.2: MVP gateway hardware box design: i) original metal box design; ii) plastic box hardware migration; iii) MVP's waterproof gateway box.

be stable and resistant to attacks from malicious processes on the gateway. Note that it was difficult to simulate an environment without GSM service, which meant test cases involving the lack of GSM were difficult to implement. Therefore, an assumption was made that GSM will always be available. The test cases of the gateway can be found in table B.2.

5.1.3 STO Garage Installation

The goal of the MVP's installation in the STO garage is to enable data acquisition. The acquired data will be analyzed offline for predictive maintenance research. To enable this, I installed the following at the STO garage: a) the gateway onto a hybrid bus, the vehicle node (VN); b) a server leader node (SLN), a laptop, for storing the J1939 data received from the test vehicle over the Wi-Fi network; and c) a Huawei E8372 Turbo stick, which is a USB Wi-Fi modem with a LTE data plan, near the fuel station. The root node (RN) in the MVP is a Meshcentral server hosted by *meshcentral.com* (see [65] for more information on Meshcentral), a website that hosts administration servers, which simply provides remote administration services to the SLN and VN. Figure 5.3 illustrates the MVP's setup at the STO garage.

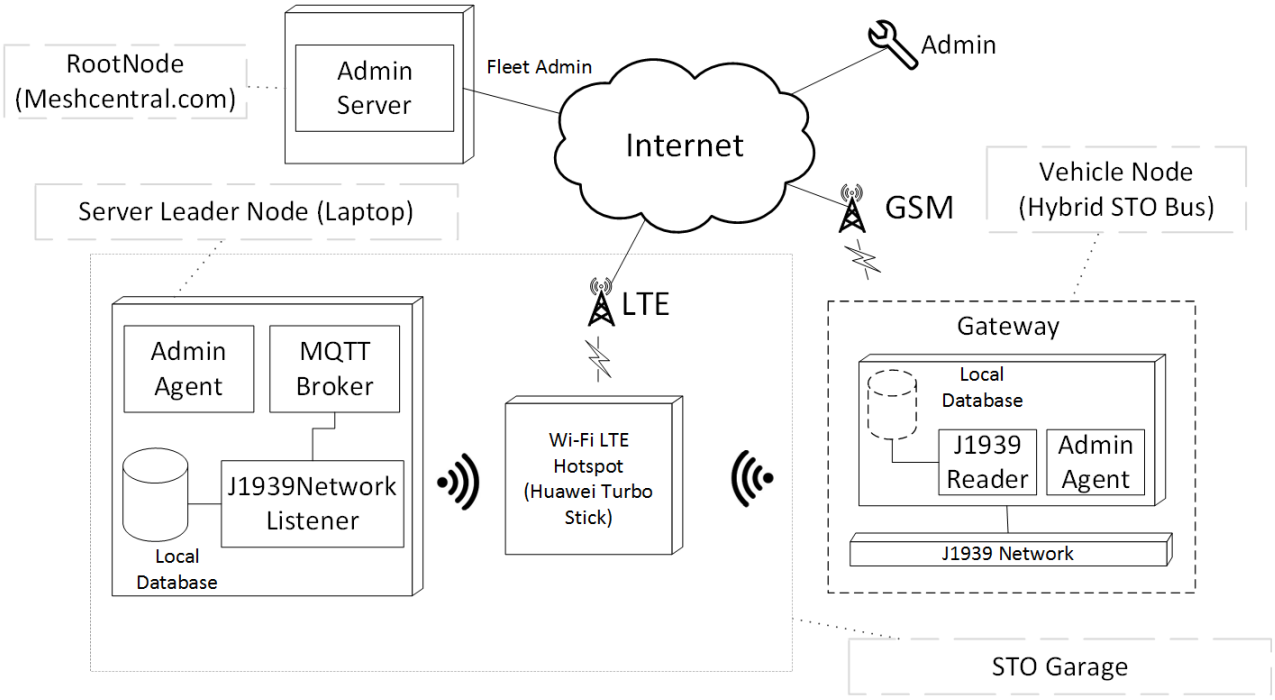

Figure 5.3: Simple diagram of the MVP architecture deployed at the STO garage.

5.1.3.1 Network Connection

LTE

The Huawei E8372 Turbo stick is connected to the internet via a LTE data plan, requiring a SIM card. It hosts a Wi-Fi hotspot so that the SLN and VN can access the internet via Wi-Fi. I did not want to intrude on the STO's network architecture, so I used LTE data instead. The Huawei E8372 Turbo stick's firewall is configured to block all incoming connections from the internet (via LTE). This way, only devices connected to the Wi-Fi can connect to the SLN.

Wi-Fi

This network configuration allows the VN to have a stable and fast internet connection near the fuel station for a short period, since the buses spend 5 to 10 minutes there a few times per day. This provides a sufficiently large timeframe for the VN to dump J1939 data to the SLN via MQTT. A data analyst can later download the J1939 data from the SLN via Meshcentral over LTE. Note that the VN can only communicate with the SLN via Wi-Fi, because of Huawei E8372 Turbo stick's firewall.

GSM

When there is no Wi-Fi, which is the majority of time, the GSM data plan provides the VN with an internet connection. The GSM connection was only used for administrative purposes such as uploading scripts, executing commands, and managing VN services. The GSM could not be used for data dumps due to resource constraints. In a production

environment, the VN will be able to communicate with its SLN at all times (via LTE, GSM, Wi-Fi, etc.). The GSM data plan is sufficient for meeting the requirements of the MVP to acquire J1939 data.

5.1.3.2 Server Leader Node

The SLN is a laptop that runs Ubuntu, a Linux distribution. The SLN hosts a MQTT broker to provide the VN with the ability to publish J1939 data dumps to the data acquisition services on the SLN. To configure the laptop the following was done: Ubuntu was updated, Meshcentral was installed, a MQTT broker was installed, a firewall was configured to only allow incoming packets from port 1883 (MQTT's port), a SLN user was configured, Java and MySQL were installed to avoid installing them over LTE in the future, and the laptop was configured to not go into sleep-mode when the lid is shut. The details of the operating system configuration can be found in section B.3.

Test Cases

The test cases for the SLN are similar to those of the gateway (see in section 5.1.2.1). However, the SLN does not have a GSM connection and therefore has no test cases related to GSM. Instead of reading from a J1939 network, the SLN has a service that listens for J1939 data dumps from the vehicle node (over MQTT). The test cases of the SLN can be found in table B.3.

5.1.4 Data Acquisition

The J1939 packets were sampled at 1KHz (once each millisecond). In other words, each time any ECU in the J1939 network would write on the CAN-bus, the gateway would receive it and store it. The gateway split the daily data dump files into smaller pieces, which made sending a file over the Wi-Fi network faster when the file was sent in pieces. Once the VN entered the garage and the Wi-Fi became available, the VN would connect to the Wi-Fi and send each of its data dump file partitions to the SLN. The SLN would reconstruct the original file by appending all the partitions together. For more details on the data acquisition laptop configuration, see section B.3.1.

Once the entire data dump was received by the SLN, it was now accessible to an analyst via the 5GB/month LTE data plan. Data could be downloaded from the SLN for offline analysis. Since deployment, the MVP allowed us to acquire J1939 data dumps daily. Approximately 1 GB of uncompressed J1939 data (200 MB when compressed) was produced by the VN daily. This data was stored on the SLN at the STO garage. With the help of the J1939 specification document (see section 2.9.3 for more details), there were various data types decoded by analyzing the J1939 data dumps, such as sensor data, the status of equipment, J1939 network routing information, proprietary data, diagnostic trouble codes, etc. There is one data dump in particular that is used to analyze and explore sensor data. The **primary J1939 data dump** that will be analyzed in the following sections was obtained from the STO city test bus on November 23 2018, which

ran for approximately 10 hours. The data dump is 662 MB, and will be referred to the *primary STO data dump* in future sections.

5.2 Sensor Data Analysis

5.2.1 Data Format

The primary data dump is a comma separated value (CSV) file, with the following header: the elapsed time in milliseconds since the bus powered on, the parameter group number (PGN) of the J1939 packet, and the eight payload bytes. Below is an example J1939 packet from the data dump:

```
27,64480,3e,69,0,30,10,0,59,ff
```

where 27 indicates the packet was read 27 ms after the start of the J1939 data acquisition, 64480 represents the PGN, and the rest is the payload (data) of the packet.

5.2.2 J1939 Specification Document Parsing

The J1939 specification document (JSD) is an Excel file containing all required information necessary to decode a J1939 packet. The version of the document used in this work is the *J1939DA - OCT 2017* (digital annex of serial control and communication heavy duty vehicle network data approved August 2017). The document also includes a lot of unnecessary information when the only requirement is decoding a J1939 packet. This is why I shrunk the JSD into a more compressed and simple form, denoted as $J\hat{S}D$. I only chose the required columns for decoding packets and converted the JSD to a CSV file. I chose $J\hat{S}D$ to be a CSV file so that a software applications could easily manipulate it. The remaining columns found in $J\hat{S}D$ are as follows:

```
PGN,Parameter Group Label,SPN Position in PGN,SPN,SPN Name, SPN
  Length,Resolution,Offset,Units,Data Range,Operational Range
```

The following is an example of a row in $J\hat{S}D$:

```
65262,Engine Temperature 1,1,110,Engine Coolant Temperature, 1 byte,1
                    °C/bit,-40 °C,°C,-40 to 210 °C
```

To parse $J\hat{S}D$, each field and its format must be understood before it can be parsed. I implemented the JSD parser in the Java programming language (see Appendix B.4.1 for the implementation details).

5.2.3 J1939 Packet Analysis

Once the JSD has been parsed, the primary J1939 data dump is ready to be decoded for analysis (see Appendix B.4.2 for implementation details).

5.2.4 Data Exploration

With all the decoded sensor values of the STO primary data dump, it was time to analyze the data. I saved the decoded sensor data in a CSV file, so that I could use the R programming language to plot sensor values over time and plot histograms. The first thing I did was identify all the sensors that existed on the bus. A sensor existed if it had at least one non-null value in the data dump. By definition, J1939 defines `0xFF` (a binary word that exclusively contains 1s) as null, so if extracting a sensor's encoded value yields a binary word with all its bits set to 1, then it is a null value and the sensor did not actually output anything. There were 632 unique suspect parameters (sensors, diagnostic trouble codes, network information, etc.) that had non-null values. There were approximately 100 million SPN readings found in the primary STO data dump. Noteworthy sensor readings found in the data dump are listed below:

- wheel speed information: axle angle and relative speed
- vehicle distance information: trip distance, total distance, and remaining distance before running out of fuel
- driver pedal positions
- engine information: speed, torque, and temperature
    - — oil information: temperature, pressure, and level
    - — coolant information: temperature, pressure, and level
- transmission fluid information: oil temperature and pressure

I decided to investigate the following statistics of each sensor:

- minimum
- maximum
- median
- interquartile range
- standard deviation
- cardinality (number of unique values)
- count
- normalized entropy

min	max	mean	std
0	101.5	100.5505	1.680087564

Table 5.1: Statistics revealing the presence of noise for an anonymous sensor.

5.2.4.1 Sensor Selection

Machine learning (ML) algorithms cannot include all features of a dataset into their model for performance reasons, so a few techniques were applied to select a subset of sensors/features from the primary STO data dump. After I performed feature selection using the strategies discussed below, there were 61 sensors selected from the 632 parameters found in the primary STO data dump.

Data Quality Issues

After analyzing the primary STO data dump, two main quality issues were identified, namely, meaningless sensor readings and noisy sensor signals. Although some sensors had non-null values, their actual sensor readings were meaningless. Many sensors reported garbage values; for example, sensors reporting ambient temperatures over 200°C. A sensor noise analysis, discussed below, revealed that some sensors had small amounts of noise and other sensors were overwhelmed with noise.

To discover that noise was present in the dataset, I applied a few analysis techniques. By analyzing the statistical behavior of the sensors, I was able to gain insight about the data. For some sensors, the standard deviation was quite small, but the maximum and/or minimum were a great distance away from the mean, which suggests outliers (or noise) are present [50]. For example, table 5.1 presents a summary of the statistics of an anonymous sensor where the presence of noise is suggested.

The extreme minimum value of 0 suggests there is noise, since the mean is 100.55 and the standard deviation is low. When such statistics suggest the presence of noise, both a time-series graph and a histogram plot can be useful to further investigate the presence of noise. Figure 5.4 reveals the presence of noise in an anonymous sensor's data stream, where the outlier/noisy values are the small values below 20 in the graph.

Once I identified the sensors with potential outliers, I visualized their behavior over time (using a time-series graph) and their data distribution (estimated using a histogram plot). These graphs were created using the R programming language.

To reduce the noise in the dataset, when possible, I excluded readings that exceeded the operational range of the sensors (when applicable), since an assumption is that the primary STO dataset includes only normal behavior data. Another approach to reduce noise would have been to use a moving average of the sensor readings.

Removing Sensors

According to [50], features with a cardinality of 1 (that is, 1 unique value) can usually be excluded from a dataset, since they are unlikely to benefit a ML model. That is, a sensor that always outputs the same value will be useless to a ML model, since when an

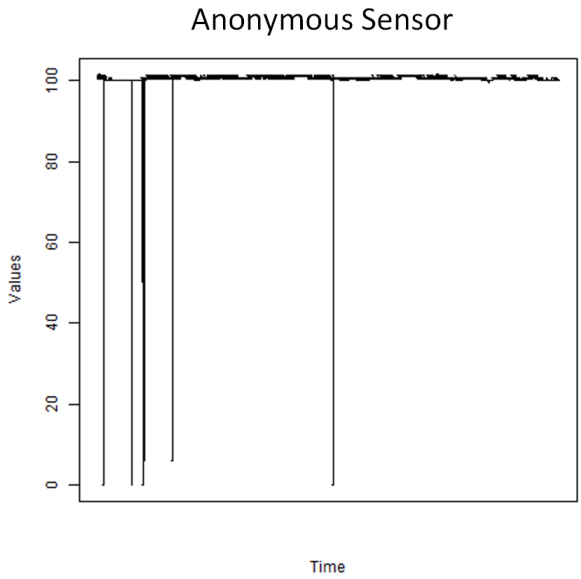

Figure 5.4: Anonymous sensor's behavior over time, with presence of outliers.

event occurs, the sensor's output stream will most likely remain unchanged and therefore, no pattern will be detected. Note that this strategy is used by [82].

I removed most of the sensors with a cardinality of less than 9 from the primary STO dataset, since many sensor signals were filled with noise. I arbitrarily chose 9, since it was a fitting choice for the dataset. The majority of the sensors with low cardinality were sensors with a cardinality of 1 and the remaining unique values were due to noise, which is illustrated in figure 5.5. In this example, a hypothetical sensor's readings were constant for the full duration, but there were the occasional noise readings. In this case the cardinality of the sensor stream is virtually 1, but due to noise, the cardinality is 8.

Another example follows: figure 5.6 depicts an anonymous sensor that reports the volume of a liquid in a container from 0 to 100%, where a reading of 0% indicates the container is empty and 100% indicates the container is full. The container is 100% full at the beginning of the trip and is slowly draining over the 10-h data acquisition period. At the end of the trip, the container is filled back to 100%. It is unclear if the cluster of data readings at the start of the trip between 95% and 100% are outliers. What is clear is that the data readings at the bottom of the graph, those between 0% and 20%, are clearly outliers. The sensor readings generally have a smooth decreasing trend. It does not make sense that the container would lose approximately 90% of its liquid in an instant, and then a moment later regain 90% of its liquid. In this case there are 5 outliers.

Since I do not have much knowledge of mechanical engineering, the above example supports my theory that most sensors have a lot of noise in the form of small values. This is why I chose to filter out sensors with low cardinality, since analysis revealed that most sensor streams with low cardinality mostly included noise.

Upon further investigation of the primary STO dataset, I discovered that there were cases where sensors had less than 9 cardinality but clearly were not overcome with noise.

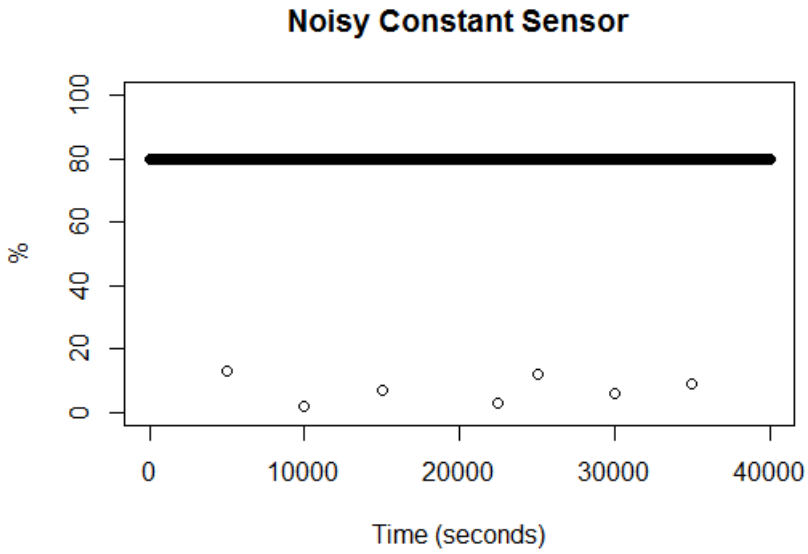

Figure 5.5: Example of a constant synthetic sensor data stream with 8 unique noise samples (virtually cardinality = 1).

Such sensors could therefore provide meaningful information and were not excluded from the dataset. I also excluded all the sensors that had the "bit" unit of measurement, since readings of those sensors/parameters were almost all related to useless J1939 routing information, commands, requests, and network statuses.

5.3 Fleet Simulation Implementation

A fleet simulation system is designed and implemented in this thesis to compare the performance of [82]'s COSMO approach and ICOSMO (see section 4.2), and to compare the performance of various histogram distance measures when used in the COSMO approach (and ICOSMO). This simulation is required, because a labeled dataset of faults and normal data is not available. That is, the primary STO dataset (discussed in section 5.1.4) is unlabeled.

In this simulation multiple vehicles are simulated, which produce synthetic data based on the primary STO data dump obtained from the hybrid bus. Faults and repairs are also generated, which affect sensor behavior. To compare the various deviation detection approaches, I used the area under the ROC curves to compare the various deviation detection algorithms.

The simulation was designed using strategies from [9]. Java was used as the main programming language to implement the simulations. The version of Java used was 1.8.0_171. An external Java library for statistical computing, *Colt* version 1.2.0, was used for randomly generating synthetic data based on arbitrary normal and uniform distributions. This

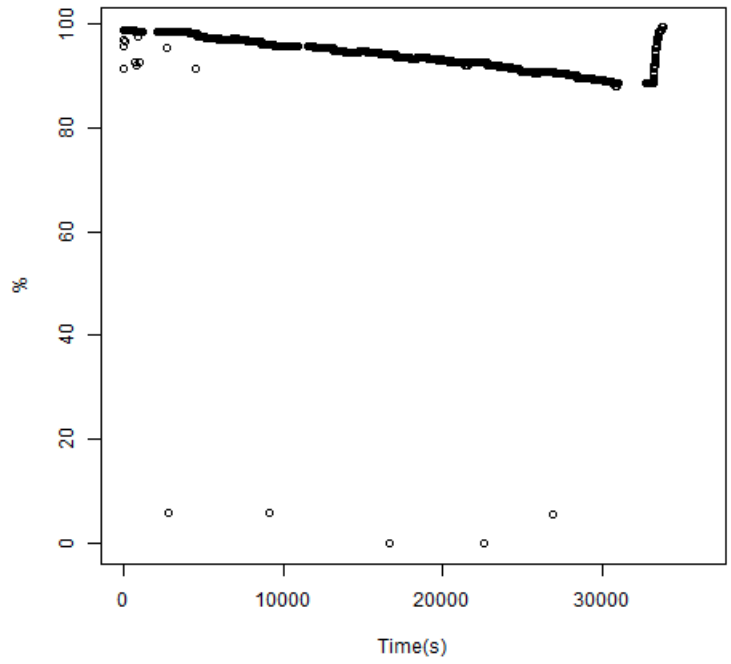

Figure 5.6: Anonymous liquid percent remaining graph.

framework was chosen, since it offers many statistical tools. To compute the ROC curves, the R programming language, version 3.6.0, was used.

5.3.1 Problem Description

The simulation is separated into three phases, namely, the configuration phase, the generation phase, and the analysis phase. The **configuration phase** loads all necessary parameters and input for the simulation. Once all the internal objects are loaded, the simulation is ready to be run. The input and parameters are found in XML configuration files. The primary STO J1939 dataset is used as the foundation for sensor data generation, which is found in a CSV file. The **generation phase** generates faults, repairs, and sensor data. The sensor data are then fed into the deviation detection models, and they produce daily z-scores for all the sensors for each vehicle. All the attributes of the sensors, such as a flag indicating whether it is a COSMO sensor, its last assigned z-score, its z-score moving average, and its sensor id, are recorded into a history of the entire simulation. The history also stores the fault and repair occurrences. The **analysis phase** analyzes the performance of the deviation detection models, by using the history created by the generation phase as a labeled dataset. This phase counts the TP, FP, TN, and FN rates by comparing all the z-scores to the threshold values and checking whether a fault occurred. It does so for all thresholds, creating ROC curve points in the process. The important elements contained in the ROC curve points include the threshold used, the TPR, and the FPR. These points are output to a CSV file, and are then processed by the R programming language to create the ROC curves. The area under each curve is then computed.

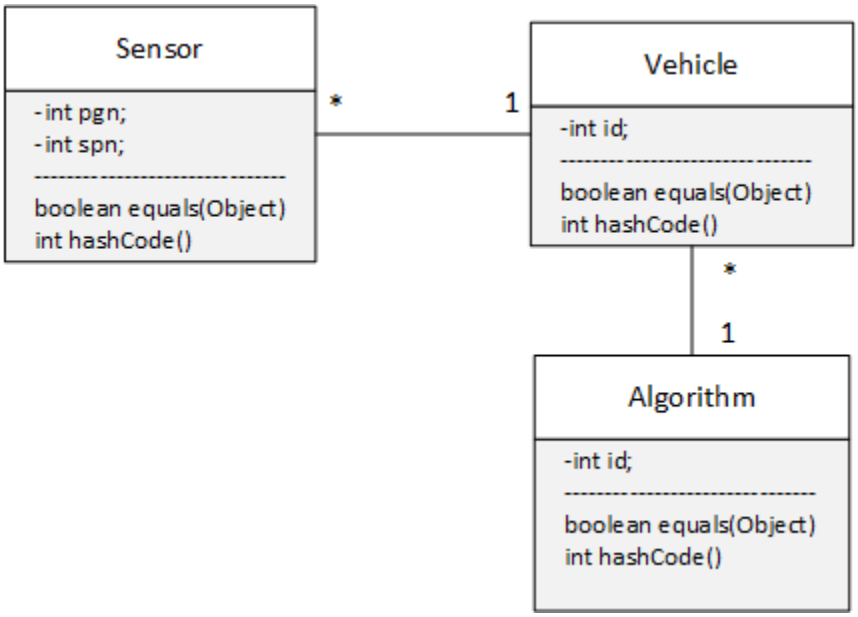

Figure 5.7: UML class diagram of the core classes used in this simulation.

It is worth noting that the time unit chosen for the simulation is discrete time. That is, time variables only have values at discrete points in time, which represent simulated days. This design choice was made, since the z-scores are computed from daily histograms.

Note that figure 5.7 illustrates the core classes/entities used in the simulation. Theses classes implement the Java `boolean equals(Object)` and `int hashCode()` functions to allow them to be used as keys in `java.util.HashMap` objects.

5.3.2 Assumptions

There is only one J1939 dataset used in the simulations, which is the one primary STO data dump obtained from the single test bus, due to time constraints of this thesis.

- **Assumption 1.** Repairs occur spontaneously, so no time is spent in the garage to repair faults. There is not enough data to replicate the garage environment when a bus is being repaired, so this assumption is made for simplicity.
- **Assumption 2.** There are no periods where buses are in the garage, to simplify the simulation model and avoid deviations due to differences in ambient conditions of the garage. There is not enough data to replicate the garage environment.
- **Assumption 3.** The faults that occur on buses do not have overlapping on-board fault-involved sensors. That is, no two faults on the same bus can occur that share a subset of fault-involved sensors. It is a simplification, since it may be the case that faults overlap when affecting sensor behavior, but no fault data is available to accurately simulate this.

5.3.3 Definitions

- **Entity** is a permanent physical object that is being modeled by the simulation (a vehicle, for example)
- **Transient Entity** is an entity that enters and leaves the model (a fault, for example)
- **Input Entity Streams** model the arrival and the departure of entities (for example, vehicles that enter a garage and then leave)
- **Output Entity Streams** model the departure and the arrival of entities (for example, vehicles that leave a vehicle manufacturing plant and are later purchased (arrive for another entity; for example, a car dealership))
- **Constant** remains the same during the simulation and all experiments. For example, the value of π, or the J1939 definition of a sensor
- **Parameter** is varied and manipulated to analyze the effect on results. For example, the number of vehicles in a fleet or a user-defined threshold
- **Time Variable** is a variable $X(t)$ that its value is dependant on the time
- **Input Variable** is a variable that originates from the outside of the environment, affects the simulation, and is not affected by the state of the simulation
- **Output Variable** is a variable that is affected as time progresses. When time changes, the value of the output variable and a timestamp are output
- **Derived Scalar Output Variable** are the output variables such that the sequence used to obtain the aggregated value is not important, but the result is. For example, minimum, maximum, and average values.

5.3.4 Random Number Generation

In the simulation there are many random events that are generated. The way I generate random events is using a p-value and a random number sampled from a uniform distribution between 0 and 1. From experimental results, Java's random number generation library meets the requirements for generating numbers between 0 and 1, and it is also much faster than using the Colt library. To generate many random numbers between 0 and 1, I used the `java.util.Random` and `java.util.stream.DoubleStream` Java classes. Instances of the `Random` class provide unique random seeds which are as good as the Java Virtual Machine (JVM) can provide. The method `Random.doubles(int size)` provides a `DoubleStream` of *size* random numbers between 0 and 1. Although this is sufficient for generating many numbers between 0 and 1, it does not meet the requirement for generating random samples from other distributions. The *Cern* library is used in this case.

In the case of the *Cern* library, the `cern.jet.random.engine.DRand` class is used to create instances of `cern.jet.random.Normal` and `cern.jet.random.Uniform`, which are

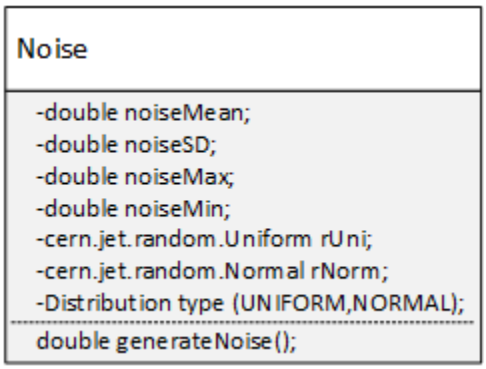

Figure 5.8: UML class diagram of the noise class, which is used for generating random numbers.

used to generate random numbers. However, the default constructor uses a constant default seed, which means any object of this class built from default constructor will have the same seed used to generate random samples. Therefore, care must be taken to avoid generating multiple identical random streams. When using instances of the `DRand` class, I used the current time as the seed, and ensured the threads creating such objects slept for random amounts of time to guarantee the `DRand` instances did not share the same seed.

I defined a `Noise` class, which is illustrated in figure 5.8 in UML, to generate random numbers. This is the class that uses *Cern*'s API to generate numbers using the attributes found in `Noise` objects. The `Noise.generateNoise()` method generates random samples from the desired distribution, using the *Cern* library, which is shown in Appendix B.5.1:

5.3.5 Configuration Phase

Problem Statement This module loads all the input and parameters into the simulation. Configuration XML files are used to store input and parameters. The simulation is given a configuration file path, reads it, parses it, and uses its entries to load various other simulation resources into memory. This phase starts the simulation.

Input XML configuration file path are input into the system.

Methodology Instances of the Java class `java.util.Properties` are used to access the XML entries from the input file. It expects an XML file of the format found in Appendix B.5.2.1:

The `Properties` class provides the basic API for accessing the XML entries. Objects of this class expect key-value pair entries as shown above, where accessing the values in the XML file is done using the keys. It does not support embedded elements. Accessing a value found in a configuration file in Java is illustrated in Appendix B.5.2.2:

However, some Java objects are complex and require many input and parameters that would require sub entries to be found in the XML file. I decided to represent complex objects by having multiple configuration files linked together. That way, the input required

can be complex and is not restricted to only key-value pairs and a directory of configuration files, as well as sub directories, can be used to store complex object associations and relationships.

For example, let a `ClassRoom` object have many registered students. If only one `ClassRoom` object is required, it would be sufficient to define entries for each student as shown in Appendix B.5.2.3, using a single configuration file.

However, this does not support multiple `ClassRoom` objects. For example, in the case that a `School` object with multiple `ClassRoom` objects needs to be loaded into memory, multiple `ClassRoom` configuration files can be used to represent each complex object. A `School`'s configuration file entries can then be file paths to `ClassRoom` configuration files, which logically represents object associations. The `School` configuration file can be structured as shown in Appendix B.5.2.4.

This configuration file design choice allows any type of object to be represented by XML files by recursively loading configuration files into memory. This is the approach taken for loading all complex objects in the simulation.

5.3.5.1 J1939 Sensor Data

Problem Statement I use the primary STO data dump as the real J1939 sensor data in all the simulations. Details for processing J1939 sensor data dump is explained in section 5.1.4. This module parses the J1939 specification document (JSD) and decodes all the sensor readings from the STO data dump. This module provides the other simulation modules with a global reference to the primary STO dataset.

Input J1939 data dump file path f. String that represents the file that contains the primary STO J1939 data dump of J1939 packets. J1939 specification document file path j. String that represents the JSD CSV file.

Output A raw sensor data output entity stream, O, is the output. Where each reading is paired with noise, where the noise is uniformly distributed between the maximum and minimum values of the sensor defined in the J1939 specification document. So an entity output to O has the form {{`pgn,spn`},`sensor value,noise sample`}. In the implementation, I used a `java.util.HashMap` with the following form for the output stream `HashMap<Sensor,List<ValueNoisePair>>` , where `ValueNoisePair` is a wrapper class with two double attributes, namely, *value* and *noise*.

Parameters A white-list of sensors W to include in the sensor data output, which is used to avoid reading undesired sensors, that is, only desired sensors' data are output. There are many unnecessary PGN-SPN pairs for the simulations, since some pairs do not represent sensor readings. For example, some pairs are J1939 routing information such as the source address of a packet. The details on how the sensor white-list is compiled are discussed in section 5.2.4.1.

Filter probability p-value, f_p, a sampling parameter. Since there is approximately 600 MB of encoded J1939 sensor data, it would slow down the simulations considerably if all the data was included in the simulation. $100 \cdot f_p$ represents the fraction, in percentage,

of readings to filter from the dataset, and therefore $1 - f_p$ is the fraction of readings to include.

Methodology

Configuration To configure this module, all the sensor descriptions of W, from the JSD, need to be loaded into memory before sensor readings can be processed. The JSD is parsed into `SensorDescription` objects, which have the attributes discussed in section 5.2.2, and are stored in two `java.util.HashMap` objects, with the following form: `HashMap<PGN,HashMap<SPN,SensorDescription>>`, where PGN and SPN are of the class Integer. This allows for quick sensor description lookup when given the PGN of a J1939 packet.

Simulation Logic This module only samples a fraction of the sensor readings, for simulation speed performance reasons. This module iterates the J1939 sensor readings, r, one by one. It always includes the first reading to avoid having an empty sensor reading dataset. After the first sample, it only includes a fraction of the sensor readings by generating a random value, p, uniformly distributed between 0 and 1. If $p < (1 - f_p)$, then r is included in O, otherwise r is ignored. This way, there will be approximately $(1 - f_p) \cdot k$ sensor readings in O when all the readings have been processed, where k is the total number of sensor readings.

For each sensor reading r included in O, a random noise sample, n, is paired with r when included in O. The distribution of n depends on the minimum and maximum of the current sensor reading's sensor description.

Sensor Noise Injection Since the synthetic sensor data generation will have injected noise, I chose to generate the noise at this stage. Generating noise at this stage means that all the sensor instances will share the same noise samples when synthetic data is being generated. This means that if the synthetic noise injection is 100% for each sensor instance, then all sensors will output identical noise. However, this statistical noise bias is only the case when all the sensor behaviors' base/injected noise amplification factor, β, attributes are the same (see equation 5.1). To remove the bias, β can be a random variable so that each sensor instance will most likely produce different noise (this is discussed in detail in section 5.3.6.2). Additionally, since each sensor instance has a different probability of injecting a noise sample into its synthetically generated distribution, the sensor instances will not necessarily share the same noise even when their β attributes are equal. The advantage to this design choice is it takes considerably less time to run simulations, since the noise dataset is generated once during the whole simulation instead of each simulated day.

5.3.6 Generation Phase

This phase is the core of the simulation. It generates all the random events and other necessary output required to evaluate the performance of COSMO and ICOSMO when using a variety of histogram distances. During this phase, faults and repairs will be generated, as well as synthetic sensor data, and COSMO will be used to produce z-scores for each sensor instance in the simulation.

5.3.6.1 Fault Generation Model

Problem Statement This model is responsible for generating faults and repairs. The entities involved in this model are Fault, Repairs, Sensors, and Vehicles. Faults are transient entities that occur on vehicles, leaving the model when repaired. They define how sensor behavior will be affected when they occur. They either change/modify fault-involved sensors' current behavior or define an entirely new behavior. Repairs are transient entities that occur to remove existing faults on the vehicles. Faults and repairs have their own daily probability of occurrence for each permanent vehicle entity. Sensors are permanent entities affected by faults.

When faults occurs on a vehicle, a list of affected sensors and the fault's sensor behavior is output. A fault cannot occur more than once before it is repaired. Once repaired, a fault can occur again. When a repair occurs, a list of sensors affected by the fault repair is output, and the repair and fault leave the model. A repair cannot occur if the fault it repairs has not occurred yet on a vehicle. A repair cannot occur on the same day of the fault, since the fault should at least have one day of changing the behavior of the fault-involved sensors. Realistically a fault would not immediately be fixed, so sensors would have time to report anomaly behavior for a period of time. There is a minimum number of days a fault will persist before the repair occurs. This minimum is defined as a parameter. Each type of fault will have the same minimum, to simulate the nature of finding faults. That is, regardless of the vehicle, each type of fault will have a minimum time before being discovered, to emulate the repair process of mechanics searching for faults on vehicles. Therefore, some types of faults can be defined to be easier to find than others. For example, an overheating engine would arguably be easier to find than a faulty fan sensor that reports readings 10% higher than the norm.

There are a variety of faults that can occur. The type of faults will be constant throughout the simulations. Each has varying probability of occurrence and repair occurrence for different vehicles. Each day that passes will potentially cause new faults to activate or repairs to occur on various vehicles. A fault is associated to a `FaultDescription`, which describes a type of fault. `FaultDescriptions` define how they affect the output behavior of a list of fault-involved sensors, if a fault of that type would occur. The set of `FaultDescriptions` remains constant for the during of a simulation.

Input The input variables in this model is the time t, indicating the day. This variable is used to keep track of the virtual time.

Output The model has two output variables, namely an output entity stream of repairs RO, and an output entity stream of faults FO. These streams will be lists of repairs and faults that occur on day t. In other words, the entities in FO and RO are of the form of a (vehicle id, `FaultDescription`) pair. This way, other modules can determine which vehicle and what fault occurred at time t.

Parameters There are a few parameters in this model. n is defined as how many `Vehicle` entities will be created in the simulation. FD is the set of `FaultDescriptions`. FG is the set of fault probability generators (distribution parameters). Each generator fg in FG is used to generate the fault probabilities of a `FaultDescription` for all vehicles.

Referring to figure 5.9, the *occurrenceProb* and *repairProb* p-value attributes of the `Fault` objects (of type FaultDescription i) are created by sampling from FG_i.

m is a probability pair matrix. A probability pair is defined as (f,r), where f is the probability the fault occurs when a new day occurs, and r is the probability of repairing the fault, given that f has occurred in the past. The rows of m represent each vehicle, and the columns represent `FaultDescriptions`. Therefore, a cell at row i and column j is the probability pair for `Vehicle` i of `FaultDescription` j, which represents the probability a `Fault` of type j will occur on `Vehicle` i. Another parameter is used to define the minimum number of days faults will persist before being repaired.

Fault Descriptions A `FaultDescription` describes a type of fault. This type of fault has a set of sensor behaviors that define the effect the fault will have on some sensors P when a fault of this type appears. A `FaultDescription` also contains the set of sensors $\overline{P}$ that are not fault-involved; that is, are not affected when a fault of this type occurs. `FaultDescriptions` explicitly have P defined, while $\overline{P}$ is implicitly defined as $\overline{P} = S - P$, where S is the set of all sensors found in the simulation, the sensor white-list W (see section 5.3.5.1).

There is a `FaultDescription` for all the sensors to simulate a faulty sensor. These faulty sensor `FaultDescriptions` add noise to the normal sensors' behavior. Although the faults are not designed using expert knowledge, the purpose is to change the distribution of a set of sensors, which should be detected by the deviation detection models, so it should be fine that the faults are not entirely properly designed. I defined 20 `FaultDescriptions` that combined multiple sensors; for example, *Run-away Engine Fan*, where the sensor behavior is affected such that the engine fan adopts a new behavior where it is always running at 100% RPM and the engine temperature sensor is modified to be approximately 100°C cooler than normal. Table 5.2 illustrates the sensor behavior attributes required to simulate this fault, where the *NEW* behavior type indicates that the sensor behavior of the target sensor is overwritten and *MODIFY* behavior type indicates the target sensor's behavior attributes will be modified (either subtracting or adding the target behavior's attributes).

Methodology Figure 5.9 illustrates the UML class diagram of the classes used in the fault generation model.

Configuration Create the set of n Vehicles V, each with a unique id according to its creation index; that is, the ith created vehicle has id = i. Iterate all the fault probability generators $fg \in FG$. Let fd(fg) denote the `FaultDescription` of fg. Let min(fg) denote the *minDasyBeforeRepair* attribute of fd(fg). Let Fnoise(fg) be the `Noise` object (discussed in section 5.3.4), which is used to generate the probability of fault occurrence p-value, and Rnoise(fg) be the `Noise` object used to generate the probability of repair occurrence. For each $v \in V$, create a `Fault` object f and assign its attributes as follows:

$$f.\text{minDaysBeforeRepaired} = \min(fg)$$
$$f.\text{faultDescription} = \text{fd}(fg)$$
$$f.\text{occurrenceProb} = \text{Fnoise}(fg).\text{generateNoise}()$$
$$f.\text{repairProb} = \text{Rnoise}(fg).\text{generateNoise}()$$

Fault-involved Sensor Behavior 1	
Sensor	Fan Speed in %
Amplification Factor	1
Noise Amplification Factor	1
Noise P-value 1	0
Noise P-value 2	0.1
Behavior Type	NEW
White Noise Max	100
White Noise Min	95
White Noise Distribution	UNIFORM
Fault-involved Sensor Behavior 2	
Sensor	Engine Temperature
Amplification Factor	0
Noise Amplification Factor	0
Noise P-value 1	0
Noise P-value 2	0
Behavior Type	MODIFY
White Noise Mean	-100
White Noise SD	15
White Noise Distribution	NORMAL

Table 5.2: Fault-involved sensors' behavior for a run-away engine fan FaultDescription.

$$f.\text{status} = \text{Fault.Status.INACTIVE}$$

Simulation Logic Once this configuration is complete, the model can begin its simulation as time progresses. The behavior of this model's logic is illustrated using a communication diagram in figure 5.10. Every time a new day occurs, the model will use a probabilistic approach to generate faults and repairs given their defined probability of occurrence in this model's `Fault` objects.

Note that a list of fault-involved sensors for each `Vehicle` $v \in V$, denoted as FIS_v, is maintained to facilitate sensor lookup to avoid having multiple active faults share fault-involved sensors on the same vehicle. I used a `java.util.HashMap` of the form `HashMap<Vehicle,HashMap<Sensor,Sensor>>` to store fault-involved sensors, which makes lookup fast.

For every `Vehicle` $v \in V$, for each of their potential faults f, generate a random number r uniformly distributed between 0 and 1. When the expression $r < f$.occurrenceProb is satisfied, the fault f will only occur if it is not currently active and $\forall s \in f$.faultDescription.affectedSensor, s is not involved in an active fault; that is, $s \notin FIS_v$. In this case, when $r < f$.occurrenceProb, a transient fault entity is created, {`v.id,f.faultDescription`}, and will be output to FO. To record that f occurred, the day the fault occurred is recorded in the attribute f.occurrenceDay and f.status is set to the enumeration type `ACTIVE`. This is done to keep track of how long faults have been active and which fault is active.

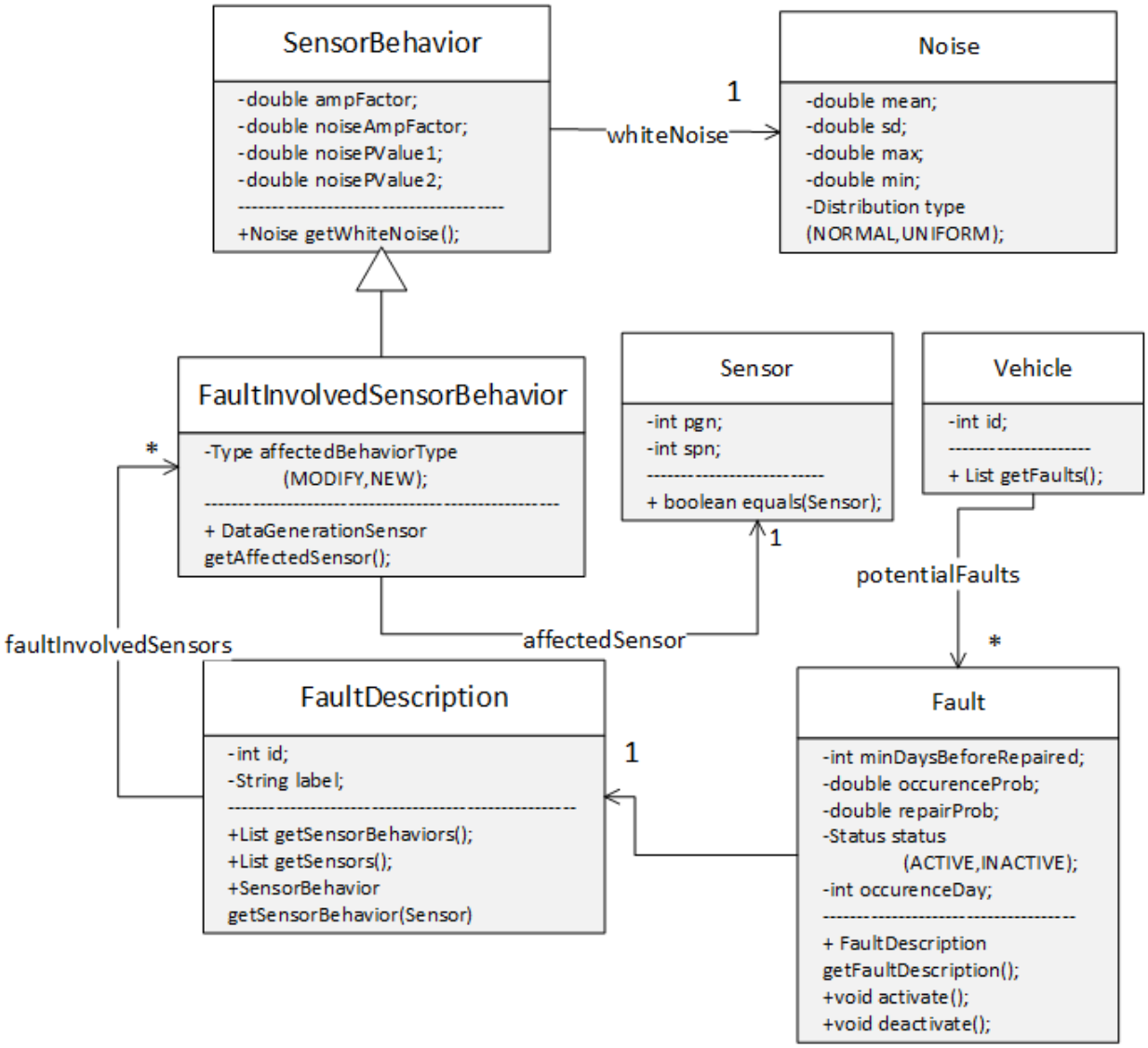

Figure 5.9: Fault generation model UML class diagram.

A similar process is done for generating repairs; that is, f.repairProb is used instead of f.occurrenceProb. The differences are that a repair cannot be created if f has not occurred yet, and additionally a repair cannot be created if the f has not been active for a sufficient amount of time. Therefore, a repair, {`v.id,f.faultDescription`} is output to RO if and only if $r < f$.repairProb and f.status = `ACTIVE` and $(t - f$.occurrenceDay) $\geq f$.minDaysBeforeRepaired. In the case of a repair occurring, the attribute f.status is set to the enumeration `INACTIVE`, and f's fault-involved sensors are removed from FIS_v, since those sensors are no longer involved with a fault on vehicle v.

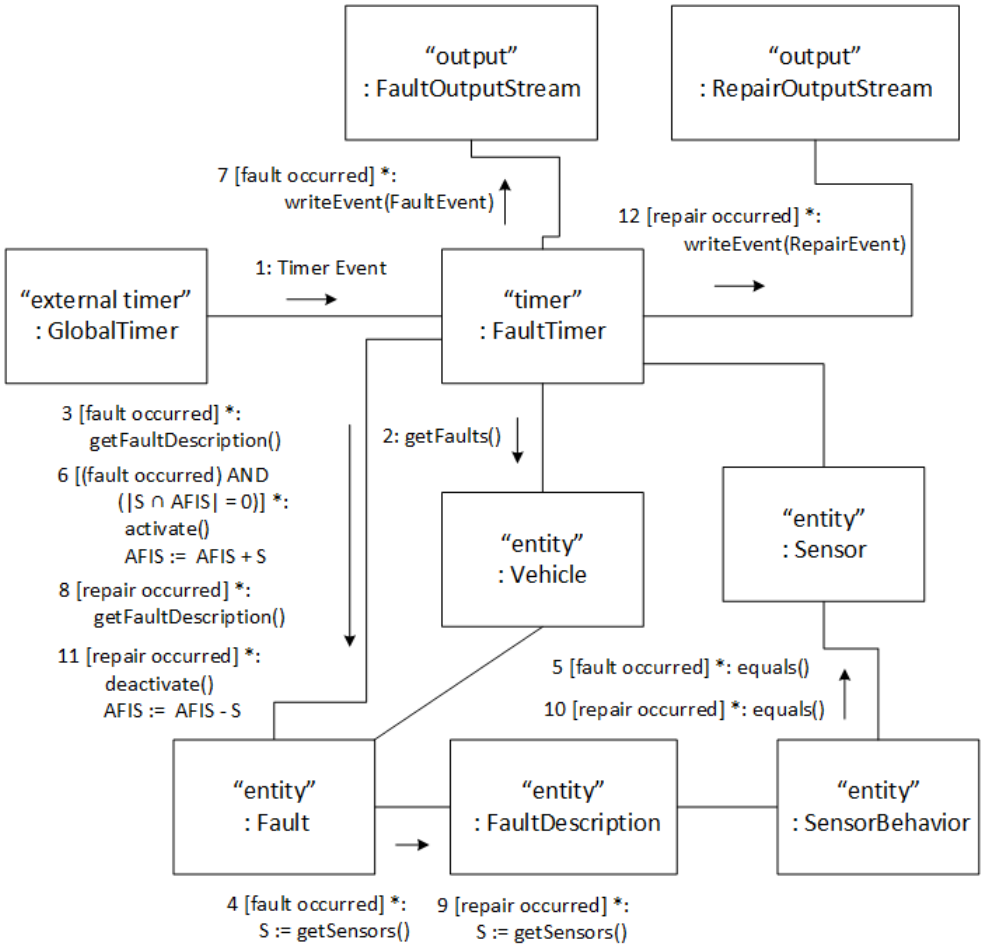

Figure 5.10: Fault generation model communication diagram, where $AFIS$ = active fault-involved sensors.

5.3.6.2 Sensor Data Generation Model

Problem Statement This model generates synthetic sensor data for a number of simulated vehicles based on the primary STO J1939 data dump. Each vehicle will be different and have a bit of noise in their sensor data to differentiate them. A stream of faults and repairs will be input into the system and affect the behavior of the sensor data generation.

Input Time t, indicating the current simulation day. Let Γ be the input entity stream of noise-value pairs for each sensor class, which is the output entity stream produced from the module discussed in section 5.3.5.1. Let Γ_i denote the set of noise-value pairs for sensor class i, where i is of the form {pgn,spn}. Let Γ_{ij} denote the jth noise-value pair of sensor class i. Let $D_{ij} = \Gamma_{ij}$.value and $N_{ij} = \Gamma_{ij}$.noise, where Γ_{ij}.value is the jth value of the STO J1939 sensor class i's reading, and Γ_{ij}.noise is the jth noise sample for the sensor class i. Two input entity streams of repair entities R and fault entities F, which indicate sensor behavior to be affected at time t, are input to this model (R and F are the output of the model discussed in section 5.3.6.1).

Notation	Meaning
m_{sid}	Sensor class id of the form {pgn,spn}
$m_{(x,\sigma)}$	Standard deviation of the `Noise` object used to generate attribute x (*)
$m_{(x,\mu)}$	Mean of the `Noise` object used to generate attribute x (*)
$m_{(x,max)}$	Maximum value of the `Noise` object used to generate attribute x (**)
$m_{(x,min)}$	Minimum value of the `Noise` object used to generate attribute x (**)
$m_{(x,D)}$	Type of distribution of the `Noise` object used to generate (uniform or normal) of attribute x

Table 5.3: Notations used to denote the inner elements of m, where $m \in M$. (*) denotes a parameter used when $m_{x,D}$ denotes a normal distribution, and (**) denotes, when $m_{x,D}$, a uniform distribution.

Notation	Meaning
β	Base/injected noise amplification factor
α	Sensor reading amplification factor
ω	Additive white noise
p_x	Probability of sampling from base/injected noise, when time t % 2 = x (% = modulo)

Table 5.4: Notation for the SensorBehavior class attributes.

Output Synthetic sensor data entity output stream O. Entities in O are of the form {vehicle.id,pgn,spn,value}.

Parameters Number of vehicles generating data, n. Set of `SensorBehavior` attribute generators, M, where each element $m \in M$ is a set of `Noise` objects (which are discussed in the random number generation part of section 5.3.4) used to generate the attributes of the n `SensorBehaviors` for each sensor class. Note that M is similar to the FG parameter discussed in the fault generation model in section 5.3.6.1. The `Noise` objects of the elements of M are referred to using the notation shown in Table 5.3.

Methodology The class diagram of this model is illustrated in figure 5.11.

Configuration

Note that the `SensorBehavior` class attributes are referred to using the notation shown in table 5.4.

Generate the set of n `Vehicle` entities with increasing ids, denoted by V. $\forall v \in V, \forall m \in M$, create a `DataGenerationSensor` entity s where the id of $s = m_{sid}$, and add s to the list of `DataGenerationSensors` of v. The `SensorBehavior` b of s must be created and assigned to the attribute s.normalBehavior. To obtain the value of the attribute b.ampFactor, a random sample $\hat{x}$ is generated, using a `Noise` object, from either a normal distribution with mean = $m_{(\alpha,\mu)}$ and standard deviation = $m_{(\alpha,\sigma)}$ or a uniform distribution with maximum value = $m_{(\alpha,max)}$ and minimum value = $m_{(\alpha,min)}$, which is determined by the distribution defined by $m_{(\alpha,D)}$. Once $\hat{X}$ is generated, b.ampFactor = $\hat{x}$. A similar process is done for β and the p_x attributes.

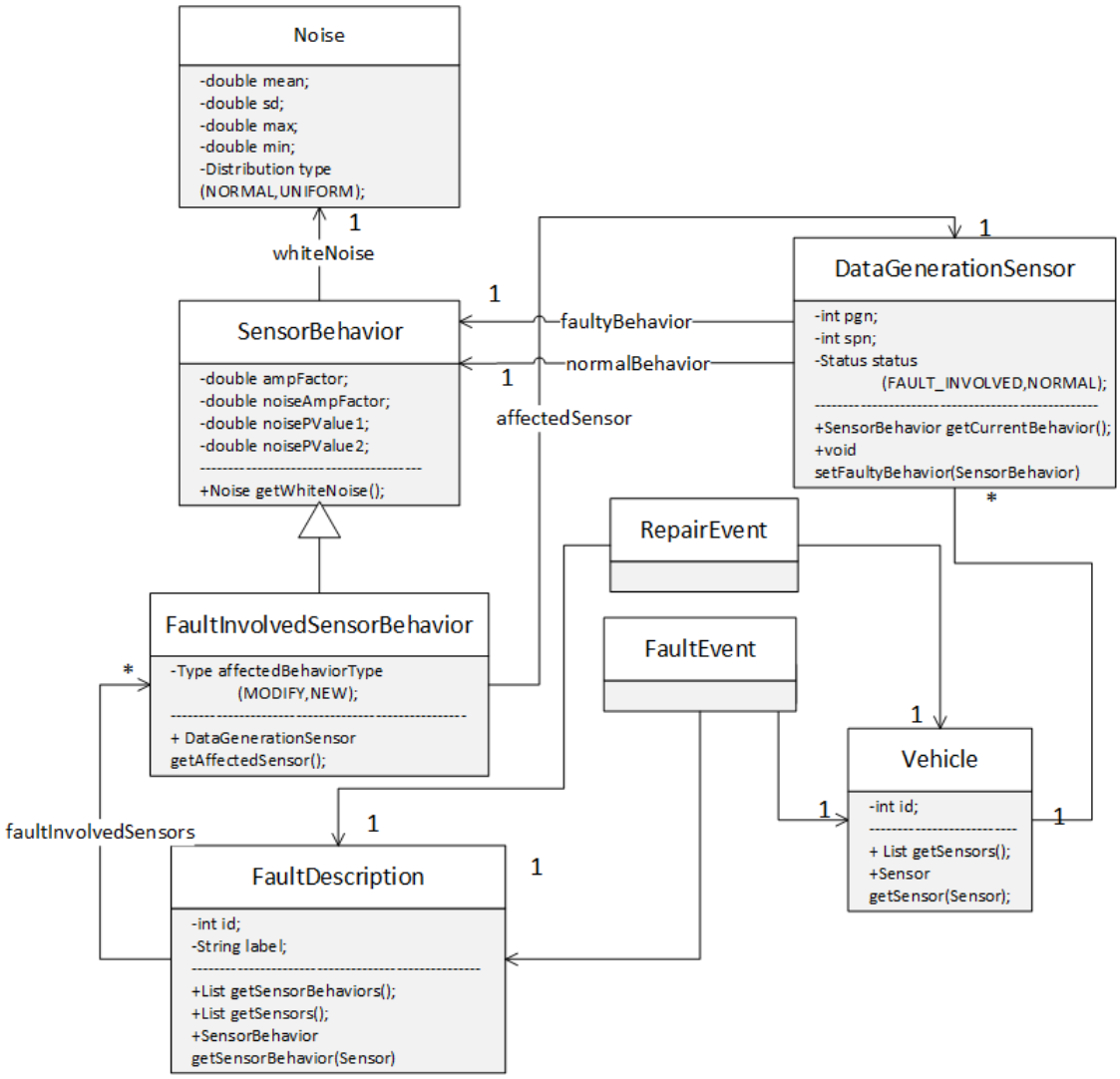

Figure 5.11: Sensor data generation model UML diagram.

The method for configuring the attribute b.whiteNoise is a bit different. Instead of generating numbers and assigning them to the attributes of b.whiteNoise, the white noise attributes found in m are directly assigned to the b.whiteNoise's attributes. In other words, b.whiteNoise.mean $= m_{(\omega,\mu)}$, b.whiteNoise.sd $= m_{(\omega,\sigma)}$, b.whiteNoise.max $= m_{(\omega,max)}$, b.whiteNoise.min $= m_{(\omega,min)}$, and b.whiteNoise.type $= m_{(\omega,D)}$. This design choice was made, since additive white noise samples will be generated for each sensor reading when a vehicle is generating a synthetic sensor data, instead of being static like the other `SensorBehavior` attribute. This should provide a desired slight variation between vehicles. .

Simulation Logic

Note that the simulation logic is illustrated using a communication diagram in figure 5.12. Each day, the time t changes, which will be used to generate daily data for each of the vehicle sensors. When t becomes $t + 1$, a new day occurs and this process is repeated.

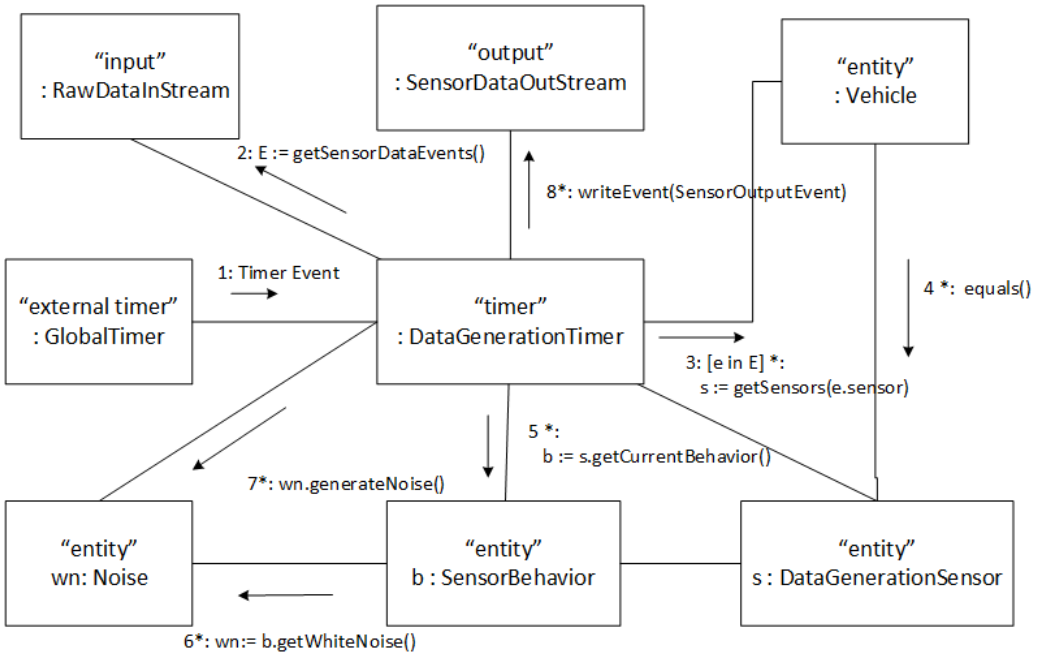

Figure 5.12: Sensor data generation model communication diagram, when generating synthetic data.

Synthetic Sensor Data Generation The communication diagram that illustrates synthetic data generation is found in figure 5.12. To generate sensor readings for all sensor classes $\hat{s}$ in Γ for each vehicle v in V, iterate all $\hat{s} \in \Gamma$. For each sensor class iterate all noise-value pairs $\Gamma_{\hat{s}k} \in \Gamma_{\hat{s}}$. For each vehicle $v \in V$, retrieve the sensor behavior b for the sensor instance $s = (v,\hat{s})$ and apply the synthetic sensor data generation equation 5.1 for all noise-value pairs to generate the synthetic data sample $\hat{\theta}_{v\hat{s}k}$, and output $\hat{\theta}_{v\hat{s}k}$ to O as the following tuple: $\{v,\hat{s}, \hat{\theta}_{v\hat{s}k}\}$; for the kth sensor reading of sensor class $\hat{s}$ of vehicle v, $\alpha_{vs} = b$.ampFactor, $\beta_{vs} = b$.noiseAmpFactor, and ω_{vs} is a random number sampled from the distribution defined by b.whiteNoise. p_x is a p-value representing the probability that noise, N_{sk}, will be sampled instead of the actual sensor reading, D_{sk}. The p-value p_x is a time variable that provides control over the synthetic sensor's signal stability. Multiple p-values are used (in this case two, p_0 and p_1), to have control over the stability of the signal (for more details on stability see section 2.15.4.4). A stable sensor signal will have small difference, Δp_x, between its p_x p-values, $|p_1 - p_0|$, while an unstable sensor signal will have a larger Δp_x. In other words, the synthetic sensor data generation can be set to be unstable by having a small portion of injected noise be found in the synthetic data on days where $t \ \% \ 2 = 0$ and a much higher proportion on days where $t \ \% \ 2 = 1$. For experimental results on the effects of the p_x p-values, see section 5.5.

r is a random number ranging from 0 to 1, sampled from a uniform distribution, and is used to decide whether to sample from the injected noise distribution or from the actual

sensor readings. It is worth noting that β_{vs} amplifies the magnitude of the injected noise.

$$\hat{\theta}_{vsk} = \begin{cases} r < p_x, & \beta_{vs} \cdot N_{sk} \\ r \geq p_x, & D_{sk} \cdot \alpha_{vs} + \omega_{vs} \end{cases} \tag{5.1}$$

Handling Faults and Repairs The communication diagram that illustrates handling faults and repairs is found in figure 5.13. The faults are simulated by changing the sensor behavior attributes of the fault-involved sensors affected by the faults. By modifying the sensor behavior attributes of sensor s on vehicle v, s's synthetic data generation will be affected, which emulates the effects of a fault. The fault will either modify the current behavior of a sensor by adjusting the sensor behavior attributes (for example, simulating degradation of a sensor or increases in temperature can be achieved using this method), or the fault will assign a new behavior to the sensors by assigning new sensor behavior attributes (for example, simulating a big leak in a liquid tank could be done by assigning a volume sensor to 0%). To accomplish this, iterate all faults f in F. For each f affecting vehicle v, iterate all the fault-involved sensors s defined in f.faultDescription.affectedSensors. For each of the fault-involved sensor behaviors b_f, affect this behavior to the current normal sensor behavior, b_n, of sensor s, but save b_n's attributes before assigning b_n with new behavior. This way b_n can be restored once the fault f is repaired. The way b_f is affected to b_n depends on the attribute b_f.affectedBehaviorType, which is an enumeration type that can either have the value `MODIFY` or `NEW`. The `MODIFY` sensor behavior modifies the existing attributes of b_n by adding the attributes of b_f, which is illustrated in equation 5.2.

$$\begin{aligned}
&s_b.\texttt{ampFactor} = s_b.\texttt{ampFactor} + s_f.\texttt{ampFactor} \\
&s_b.\texttt{noiseAmpFactor} = s_b.\texttt{noiseAmpFactor} + s_f.\texttt{noiseAmpFactor} \\
&s_b.\texttt{min} = s_b.\texttt{min} + s_f.\texttt{min} \\
&s_b.\texttt{max} = s_b.\texttt{max} + s_f.\texttt{max} \\
&s_b.\texttt{noisePValue1} = \texttt{min}(s_b.\texttt{noisePValue2} + s_f.\texttt{noisePValue1},1) \\
&s_b.\texttt{noisePValue2} = \texttt{min}(s_b.\texttt{noisePValue2} + s_f.\texttt{noisePValue2},1) \\
&s_b.\texttt{whiteNoise.min} = s_b.\texttt{whiteNoise.min} + s_f.\texttt{whiteNoise.min} \\
&s_b.\texttt{whiteNoise.max} = s_b.\texttt{whiteNoise.max} + s_f.\texttt{whiteNoise.max} \\
&s_b.\texttt{whiteNoise.mean} = s_b.\texttt{whiteNoise.mean} + s_f.\texttt{whiteNoise.mean} \\
&s_b.\texttt{whiteNoise.sd} = s_b.\texttt{whiteNoise.sd} + s_f.\texttt{whiteNoise.sd}
\end{aligned} \tag{5.2}$$

The `NEW` sensor behavior overwrites the existing attributes of b_n with b_f's attributes, which is illustrated in equation 5.3.

$$\begin{aligned}
&s_b.\texttt{ampFactor} = s_f.\texttt{ampFactor} \\
&s_b.\texttt{noiseAmpFactor} = s_f.\texttt{noiseAmpFactor} \\
&s_b.\texttt{min} = s_f.\texttt{min} \\
&s_b.\texttt{max} = s_f.\texttt{max} \\
&s_b.\texttt{noisePValue1} = s_f.\texttt{noisePValue1},1 \\
&s_b.\texttt{noisePValue2} = s_f.\texttt{noisePValue2},1 \\
&s_b.\texttt{whiteNoise.min} = s_f.\texttt{whiteNoise.min} \\
&s_b.\texttt{whiteNoise.max} = s_f.\texttt{whiteNoise.max} \\
&s_b.\texttt{whiteNoise.mean} = s_f.\texttt{whiteNoise.mean} \\
&s_b.\texttt{whiteNoise.sd} = s_f.\texttt{whiteNoise.sd}
\end{aligned} \tag{5.3}$$

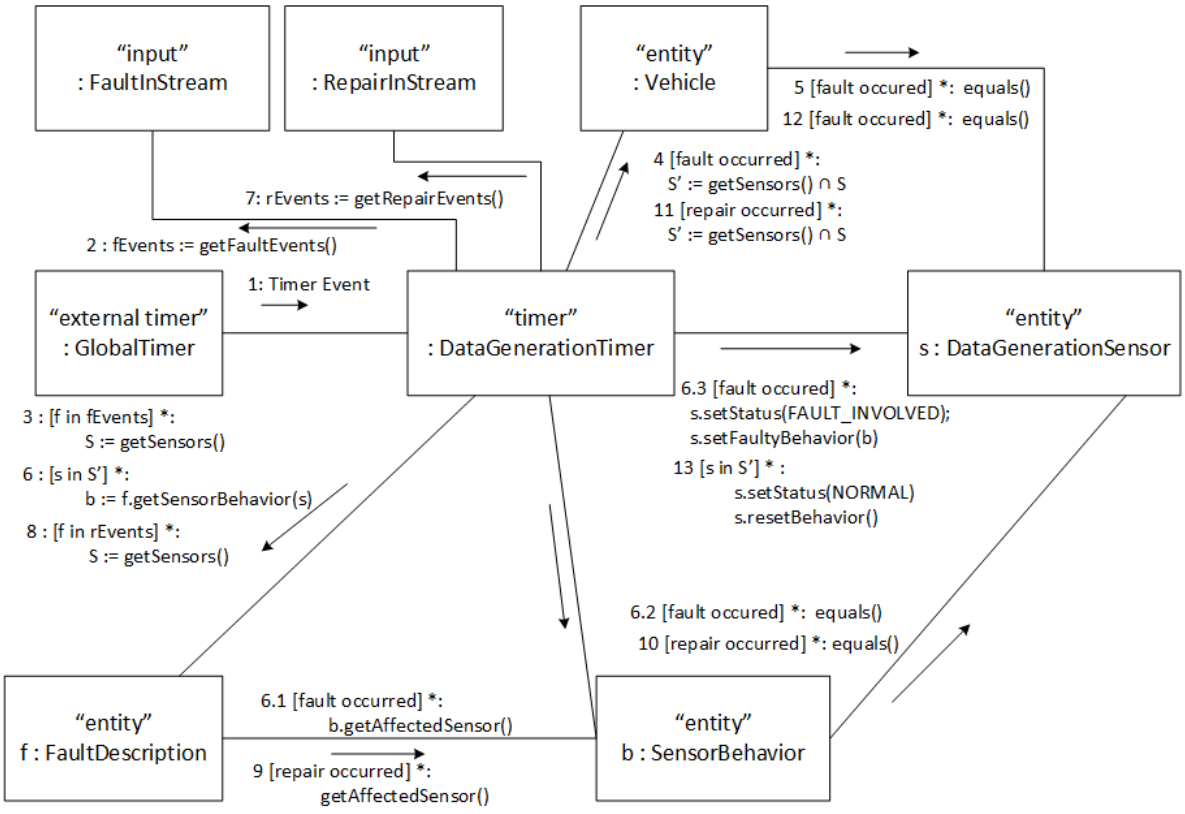

Figure 5.13: Sensor data generation model communication diagram when processing faults and repairs.

Repairs are used to restore the normal behavior of the sensors that were affected by a fault. Iterate all the repairs in R. For $r \in R$ affecting the vehicle v, iterate all the fault-involved sensors s in r.faultDescription.affectedSensors. For each s, restore its original/normal sensor behavior attributes, which were saved before being modified by the fault. This eliminates the fault by restoring normal behavior.

5.3.6.3 COSMO Deviation Detection

Problem Statement The COSMO approach [82] is partially implemented in this module. The fault diagnosis is not implemented, and the only machine learning model implemented is the histogram. The Autoencoders and linear relations are not part of the simulation in this thesis. This module performs sensor selection and deviation detection on the input synthetic sensor data output from the sensor data generation model discussed in section 5.3.6.2. A variety of histogram distance measures are used to analyze their effectiveness at detecting deviations. This module produces z-scores for all the sensor instances in the simulation. These z-scores are used by other modules to evaluate deviation detection performance.

Input Time t. Synthetic sensor data entity input stream $\hat{\theta}$, which are from the synthetic

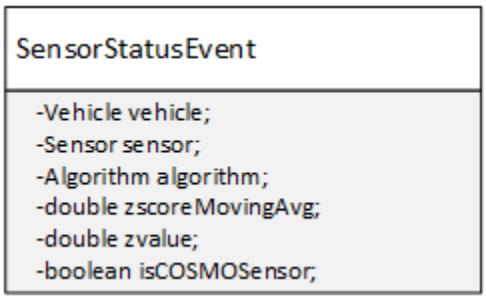

Figure 5.14: COSMO Deviation Detection Model SensorStatusEvent UML Class Diagram

sensor readings produced by the data generation model discussed in section 5.3.6.2, where $\hat{\theta}_{vsk}$ is the kth synthetic sensor reading for sensor class s on vehicle v.

Output `SensorStatusEvents` for all the sensor instances for time t, O, a `SensorStatusEvent` entity output stream. Figure 5.14 illustrates the UML class diagram of the `SensorStatusEvent` class. The attributes are explained as follows: let s denote the sensor instance that produced a `SensorStatusEvent` e, then e.vehicle is the `Vehicle` of s, e.sensor is the sensor class of s, e.algorithm is the `Algorithm` of s, e.zscoreMovingAvg is the z-score moving average window of s, e.zvalue is the last z-score produced by s, and e.isCOSMOSensor is a flag that is true when s is a COSMO sensor and false otherwise.

Parameters List of all the sensor classes in the simulation S, which is the white-list of sensors: The number of vehicles k. Number of selected sensors n to include into the COSMO approach for deviation detection. The number of daily models to put into the distance matrix, m, which is the number of days that models will be gathered before being compared and placed into the distance matrix to compute the most central pattern method. In [82] they had an m of 1 week ($m = 7$). z-score moving average window size w, which is used to decide how many z-values are considered to compute the mean to calculate the z-score. A list of Histogram distance functions F, which is a list of enumeration types used to call the appropriate distance function. When computing the stability and performing the most central pattern method, the distance enumeration type $f \in F$ indicates which histogram distance to use.

Methodology Figure 5.15 illustrates the simplified UML class diagram of this model. Note that the `SensorMap` class is actually defined as shown in Appendix B.5.3.1.

Configuration

Create a `COSMO` algorithm a for each $f \in F$, assigning f to a as its histogram distance function identifier, and let A denote this set of `COSMO` algorithms. `SensorMap` object M is created, and each $a \in A$ share a reference to the map. Create a list of k Vehicle objects V. To create the map, iterate all the algorithms $a \in A$. For each a, a) create an internal HashMap map_1 of the form `HashMap<Vehicle,HashMap<Sensor,SensorInstance>>`, b) add the key-value pair $\{a,map_1\}$ to the map M.sensorInstanceMap, and c) iterate all the vehicles $v \in V$. For each v, a) create a second internal HashMap map_2 of the form `HashMap<Sensor,SensorInstance>`, b) add the key-value pair $\{v,map_2\}$ to map_1, and c) iterate each of the sensor classes $s \in S$. For each s, create a `SensorInstance` object

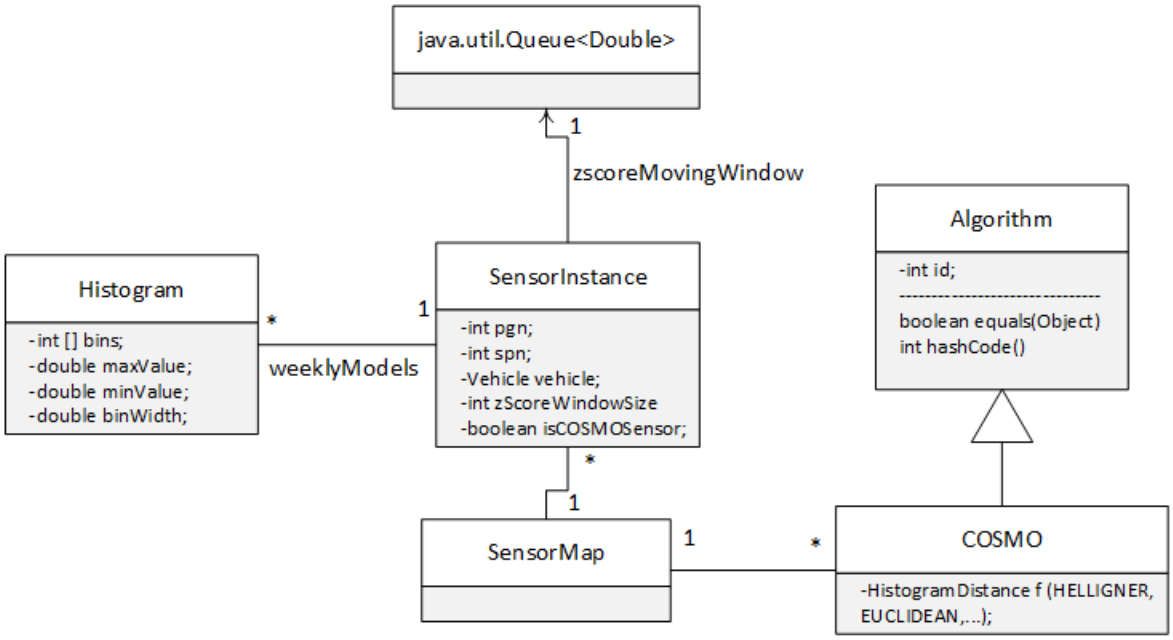

Figure 5.15: COSMO deviation detection model, simplified UML class diagram.

$i = \{v, s\}$ and add the key-value pair $\{s,i\}$ to map_2. A similar process is done to instantiate the other internal map M.sensorClassMap; however, this map shares all the sensor instance references i that were inserted into M.sensorInstanceMap. In other words, the following statement is true, assuming the `SensorInstance` at index 0 of the `SensorInstance` list in M.sensorClassMap belongs to vehicle v (see Appendix B.5.3.2 for Java source code).

Simulation Logic Each algorithm has a shared reference to a map of sensors. Each algorithm has a unique partition of the M. There are two maps in the `SensorMap` class. The M.sensorInstanceMap is used to lookup `SensorInstances` given a `Vehicle` reference. It would not be efficient to sequentially search a list of `SensorInstances`, so a `HashMap` is used, with `Sensor` class as key. Therefore, with a `Vehicle` and `Sensor` class, a `COSMO` algorithm can lookup its own unique `SensorInstance` in O(1) time, by using itself as the key lookup in the `HashMap`. The M.sensorClassMap is a map used to lookup `SensorInstances` of a `Sensor` class. At times, it is required to perform logic on all `SensorInstances` of a `Sensor` class (all `Vehicles`). Both M.sensorClassMap and M.sensorInstanceMap maps share the `SensorInstance` references, to enable flexible `SensorInstance` lookup depending on the situation.

The design choice for partitioning the `SensorMap` by `COSMO` algorithm is so each algorithm can run in its own thread, which enables concurrent execution of algorithms. Since no two algorithms will access the same `SensorInstance` reference, it requires no synchronized blocking, which makes it run much faster.

The `SensorInstance` objects in M each have a leaky queue of z-scores, with capacity of w, such that when z-scores are computed daily and added to the queue, the queue will fill up and model a moving window, since the oldest value is dropped when the queue becomes full. This way taking the mean of z-scores in the queue produces the desired z-score.

Histogram Distances Each `COSMO` algorithm created, for each element in F, has a different enumeration used to indicate which Histogram distance function is to be used. A Java `switch` statement is used on the enumeration $f \in F$ to decide which distance measure to use when comparing two histograms. The histogram distances implemented and compared in the simulations are acquired from [19], which is a survey of many histogram distance measures. I chose the Euclidean distance, since it is a common measure, and the Hellinger, since they used it in [82]. I included the Bhattacharyya distance, since [18] used it in their experiments and claimed it was efficient. The others I picked to try to have a variety of families of distance metrics to see their effects on COSMO. The goal was to analyze the performance of different histogram distance measures, to see which measure outperforms others in various simulation contexts. For example, which distance measure does well in a high noise environment. Below in equation 5.4 is a list of histogram distance measures used, where P_i and Q_i are the normalized frequency of bin i for histogram P and Q, respectively.

Distance	Equation
1. Euclidean	$\sqrt{\sum_{i=1}^{d} \lvert P_i - Q_i \rvert^2}$
2. City block	$\sum_{i=1}^{d} \lvert P_i - Q_i \rvert$
3. Hellinger	$\sqrt{\frac{1}{2} \sum_{i=1}^{N} \left(\sqrt{P_i} - \sqrt{Q_i}\right)^2}$
4. Gower	$\frac{1}{d} \sum_{i=1}^{d} \lvert P_i - Q_i \rvert$
5. Squared Euclidean	$\sum_{i=1}^{d} (P_i - Q_i)^2$
6. Intersection	$\sum_{i=1}^{d} \min(P_i, Q_i)$
7. Fidelity	$\sum_{i=1}^{d} \sqrt{P_i Q_i}$
8. Bhattacharyya	$-\ln \sum_{i=1}^{d} \sqrt{P_i Q_i}$
9. Cosine	$\frac{\sum_{i=1}^{d} P_i Q_i}{\sqrt{\sum_{i=1}^{d} P_i^2} \sqrt{\sum_{i=1}^{d} Q_i^2}}$
10. Matusita	$\sqrt{2 - 2 \sum_{i=1}^{d} \sqrt{P_i Q_i}}$

(5.4)

Histogram Number Bins

For simplicity of notation, let $\hat{s}$ denote a sensor class, N denote the number of vehicles in the fleet, $\hat{s}_i$ denote the sensor instance of class $\hat{s}$ on vehicle i, $B(\hat{s}_i)$ denote the number of bins in the histogram used to model the data stream of $\hat{s}_i$, and $B(\hat{s})$ denote the number of bins in each histogram used to model all sensor instances of $\hat{s}$.

The goal is to compute $B(\hat{s})$ such that $B(\hat{s}) = B(\hat{s}_i)$, $\forall i \in \{1, 2, ..., N\}$; however, when sensor selection is taking place, this does not hold. During the sensor selection stage, each vehicle will compute their own number of bins, using the Freedman-Diaconis rule (see equation 2.4). To compute $B(\hat{s})$, the average number of bins of the fleet is computed as follows:

$$B(\hat{s}) = \frac{1}{N} \sum_{i=1}^{N} B(\hat{s}_i)$$

Once computed, $B(\hat{s})$ is assigned to the entire fleet; that is, $B(\hat{s}_i) = B(\hat{s})$, $\forall i \in \{1, 2, ..., N\}$. Only during COSMO's sensor selection stage will two histograms, P and Q, have the possibility of having a different number of bins. Note that the distance measures in equation 5.4 assume the histograms have the same number of bins. Therefore, the following approach is taken: The distance between P and Q is calculated by assuming that the histogram with the least number of bins has additional empty bins such that P.binLength = Q.binLength. That is, let P.binLength > Q.binLength, P.binLength = k, and Q.binLength = j. Then $\sum_{i=j}^{k} Q_i = 0$. This is a reasonable approach, since the null hypothesis is that the sensor instances of the same class output samples that are from the same distribution, so they should have similar bin lengths.

Sensor Selection In general, since the sensor instances will always be generating data from the same distributions, that is, they will not be affected by external factors such as climate conditions, I perform sensor selection in two simulated days. I select the sensors based on normalized entropy (NE) and stability. To compute a sensor class's NE, I take the average NE of all sensor instances of that sensor class for both simulation days. Let $NE_d(x)$ denote the normalized entropy of some sensor instance x for simulated day d. Then the average NE for day d and $d+1$ for sensor class j can be computed by

$$\mu_j = \frac{1}{2N} \sum_{i=1}^{N} (\mathrm{NE}_d(\hat{B}_{ji}) + \mathrm{NE}_{d+1}(\hat{B}_{ji})),$$

where N is the number of vehicles in fleet and $\hat{B}_{ji}$ is the sensor instance of the sensor class j on vehicle i. The sum of the NE is divided by $2N$, since for both days each sensor instance has its NE summed.

For computing the stability, a similar process is done. Let $f(x_d, x_{d+1})$ be the distance between histogram of sensor instance x from day d and histogram of sensor instance x from day $d+1$ for some distance function f. So to compute the average stability of a sensor class j,

$$\begin{gathered} x = d \\ y = d + 1 \\ \sigma_j = \frac{1}{N} \sum_{i=1}^{N} \mathrm{f}(\hat{B}_{jix}, \hat{B}_{jiy}), \end{gathered}$$

where the sum is only divided by N, since the histograms of both days are required to compute the stability, so only N stability readings for each N vehicle are computed. Then to select the sensors, since sensor interest is defined by a combination of NE and stability,

$$\alpha_i = (1 - \mu_i) + \sigma_i,$$

where α_i range from 0 to 2 (since the stability, σ_i and average NE, μ_i, range between 0 and 1 for sensor class i) and represent the interestingness of a sensor class i. In other words, low values of α are more interesting, and high values are less interesting. This is the case since [82] describes a sensor with high NE and low stability as interesting. [82] do not define any weights associated with the measures of interestingness, so I weight both NE and stability equally.

So to perform sensor selection for all the sensor classes in the fleet, compute their interestingness and choose the top k most interesting (smallest α) sensors. Let Γ denote the set of interestingness measures for all sensor classes:

$$\Gamma = \sum_{i=1}^{M} \alpha_i,$$

where M is the number of sensor classes in the fleet. Next, sort the sensors by interest and pick the top k, where k is number of selected sensors and then sort Γ from smallest to largest. The selected sensors are those who are found in $\Gamma_{1 \text{ to } k}$, where Γ_i represents the ith interestingness measure.

Histogram Implementation

To represent a histogram, I designed a class that has an array of numbers. Each cell of the array represents the frequency (or normalized frequency) of the bin. The histogram also has three attributes, *maxValue*, *minValue*, and *binWidth*, which are used to populate the histogram when presented with samples. Note that normalized histograms are created after a histogram is finished being populated.

To populate the bins, I converted the sample value to bin index and incremented the value found at the bin index. For samples that exceed the bounds of the histogram (*maxValue* or *minValue*), they are placed in the extremity bins. For example, with a sample of -2.7 and a histogram of 0-100 bins, the sample would be placed in the 0th bin(`bins[0]` is incremented); likewise, for the same histogram, a sample of 101.3 would be placed in the last bin (`bins[bins.length-1]` is incremented). This was a design choice, since the histogram boundaries are defined by the J1939 sensor minimum and maximum values, which means no sensor value will ever exceed the bounds. The Java pseudo code in Appendix B.5.3.3 illustrates this logic.

Most Central Pattern

I implemented the MCP as described in [82]; however, I did not compute the z-scores at the same frequency as they did. In their work, they compute z-scores weekly, but for experimental evaluation purposes, I wanted z-scores to be produced daily to be able to test

daily if faults were detected or not. So, each day I replaced the oldest models with the new ones in the distance matrix and then applied the MCP method to compute the z-scores.

There is a potential source of error in my MCP implementation. The MCP method requires the most central model to be compared to all other models, which is the minimum sum row when using the Euclidean distance as a measure. However, for all histogram distance functions the minimum sum row was used to find the most central model. The minimum sum row may not be the most central model for some histogram distances. Future work would be required for investigating the nature of the most central model for a variety of histogram distance functions.

5.3.6.4 History

Problem Statement This module is responsible for recording the behavior of the entire simulation. Its role is to save the time and other important attributes of events such as faults, repairs, and sensor statuses. With a history of a simulation, the events of the simulation can be replayed, and parameters can be manipulated to see the effects they have on the output of the simulation by re-using the same history over multiple experiments. Simulation events can be accessed as time variables using a history. In other words, for some time t, the events that occurred at time t can be accessed. The history does not save each simulated sensor reading, but instead saves the daily z-scores. This module populates three time variables, H_F, H_R, and H_S, the set of faults, repairs, and sensor statuses respectively.

Input Time t, in days. Three input entity streams, namely, a `Repair` input entity stream R that contains the `Repairs` that occurred at time t, a `Fault` input entity stream F that contains the `Faults` that occurred at time t, and a `SensorStatusEvent` input entity stream S that contains the status of all the sensor instances at time t, for all the `Algorithms`.

Output Three time variables are output, H_F, the `FaultHistory`, H_R, the `RepairHistory`, and H_S, the `SensorStatusHistory` for all the time ticks of the simulation.

Methodology Figure 5.16 illustrates the UML class diagram of the important classes involved in the history module.

Configuration Three empty histories are initialized, where each is implemented using Java `HashMap` objects as the data structure of choice to record all the events and occurrence times. To configure the histories, the `HashMaps` must be partitioned, since multiple threads (one for each partition) may be accessing parts of the `HashMaps` concurrently. A partition is a section of a `HashMap` that is reserved to only be accessed by a unique thread. Each thread in the simulation has a partition key, either a `Vehicle` or `Algorithm` object, which is used to gain access to an appropriate history's partition.

Referring to figure 5.16, `K` is a Java generic type for the partition key. `Faults` and `Repairs` occur on `Vehicles`, which means the key objects used to partition the `HashMaps` for both the `RepairHistory` and `FaultHistory` are `Vehicle` objects, that is, `K` = `Vehicle`. This way, each vehicle has its own partition/section of the history. For `SensorStatusEvents`

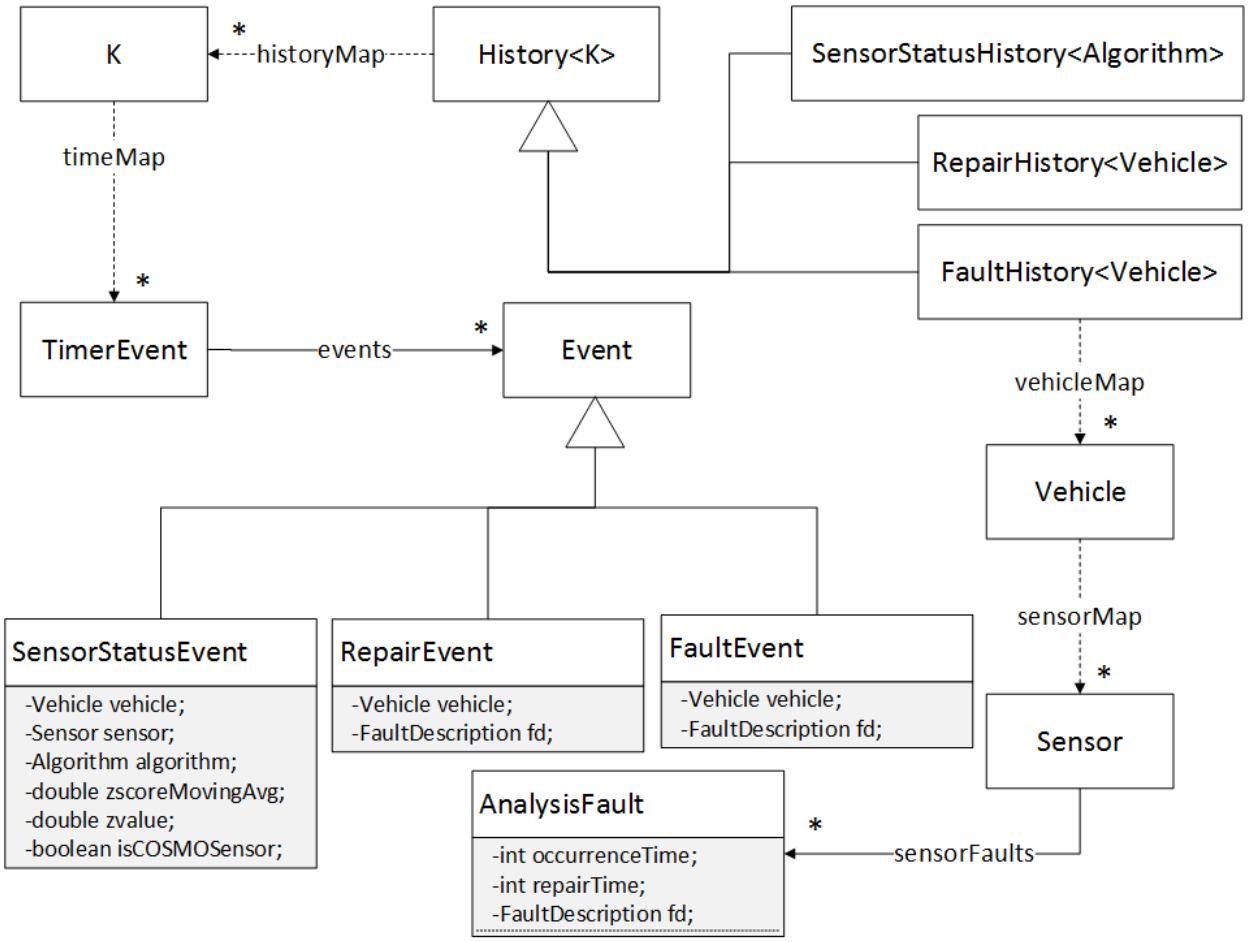

Figure 5.16: History UML class diagram. The associations represented by dashed-lines are Java HashMaps. For example, the *keys* association is of the form `HashMap<K,HashMap<TimerEvent,List<Event>>>`.

.

that are output by all the algorithms (see section 5.3.6.3), the `SensorStatusHistory` is partitioned by an `Algorithm` object. This way, each `Vehicle` and `Algorithm` have their own unique section in the histories so that the threads representing each of these objects can access the histories without using slow synchronization mechanisms.

Let H_F^v denote the partition of the `FaultHistory` reserved to the `Vehicle` v and $H_F^v(t)$ denote the set of `FaultEvent` objects that represent the faults that occurred at time t. Similarly, let $H_R^v(t)$ denote the set of `RepairEvent` objects that represent repairs that occurred at time t for the `Vehicle` v's partition, and let $H_S^a(t)$ denote the set of `SensorStatusEvent` objects that represent sensor statuses of all the sensor instances of `Algorithm` a at time t.

The histories' `HashMap` partitions are populated. That is, $\forall v \in V$, where V is the set of `Vehicles`, creates a new object o of the form `HashMap<TimerEvent,List<FaultEvent>>` and inserts the key-value pair $\{v,o\}$ into H_F's hashmap; that is, $H_F^v = o$. The same is done for configuring H_R, $H_R^v = o$, but instead $o =$ `HashMap<timerEvent,List<RepairEvent>>`.

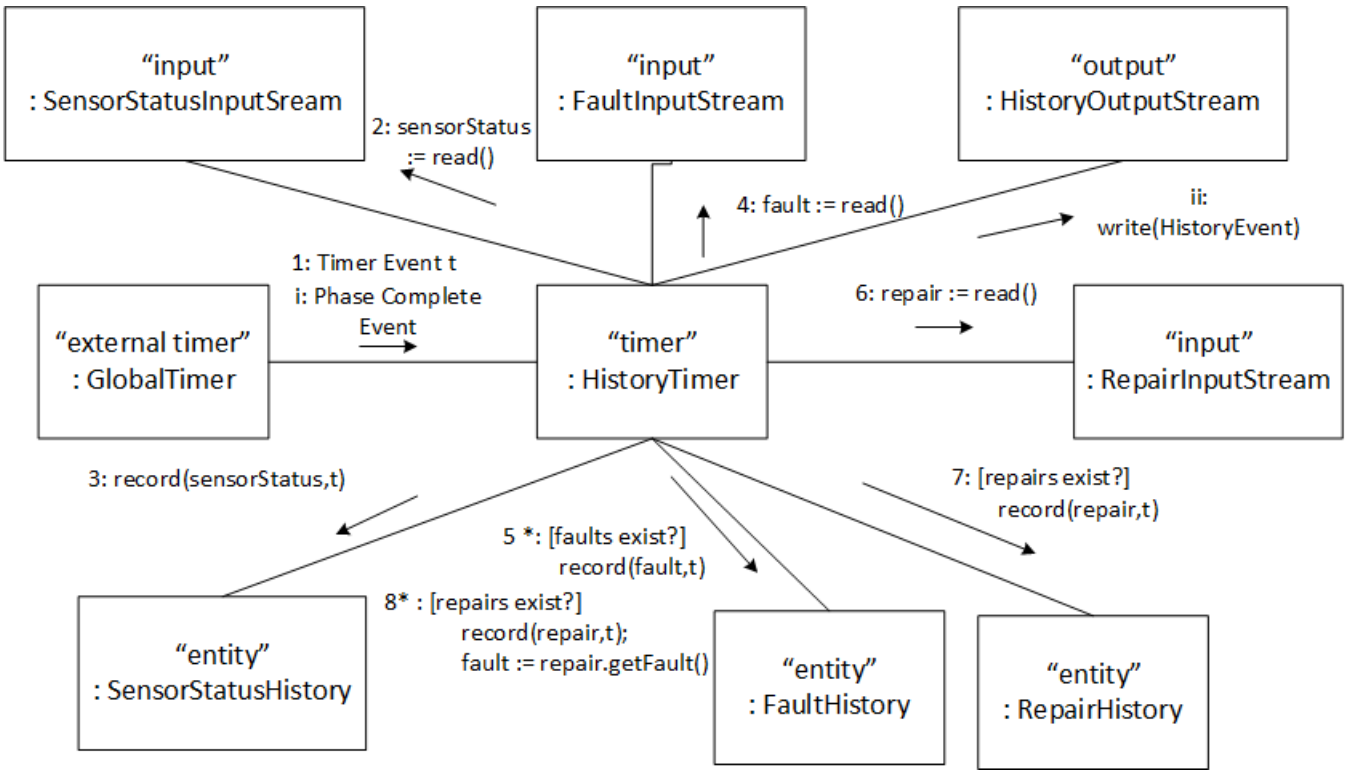

Figure 5.17: History communication diagram. Note that figure 5.18 illustrates the additional steps performed in the FaultHistory class.

For the sensor status history, $\forall a \in A$, where A is the set of `Algorithms`, $o =$ `HashMap<TimerEvent, List<SensorStatusEvent>>`, and $H_S^a = o$.

Simulation Logic Figure 5.17 illustrates the simulation logic of this module using a communication diagram.

When a new day occurs, time t, the three input streams are processed to record all their entities for time t into the appropriate history objects. To populate the `SensorStatusHistory`, $\forall e \in S$, let a denote e's `Algorithm`, $H_S^a(t) \leftarrow H_S^a(t) \cup e$. To populate the RepairHistory, $\forall e \in R$, let v denote the `Vehicle` that was repaired by e, $H_R^v(t) \leftarrow H_R^v(t) \cup e$. To populate the FaultHistory, a similar process is done for the recording faults, $\forall e \in F, H_F^v(t) \leftarrow H_F^v(t) \cup e$, where v is the `Vehicle` where a fault occurred.

There is a bit more processing involved for populating the `FaultHistory`. It is worth noting that the `FaultHistory` class has additional associations (see figure 5.16) to enable efficient fault occurrence date lookup. Not only can the `FaultHistory` be used to look up faults that occurred on a specific day, it can also be used to lookup faults that occurred for a sensor instance. As a result, when a fault is recorded in $H_F^v(t)$, there is a bit more processing involved (see figure 5.18).

FaultHistory Additional Map The `FaultHistory` has an additional `HashMap` used for storing a list of `AnalysisFault` objects that represent faults that occurred on a `Vehicle` for some fault-involved sensor.

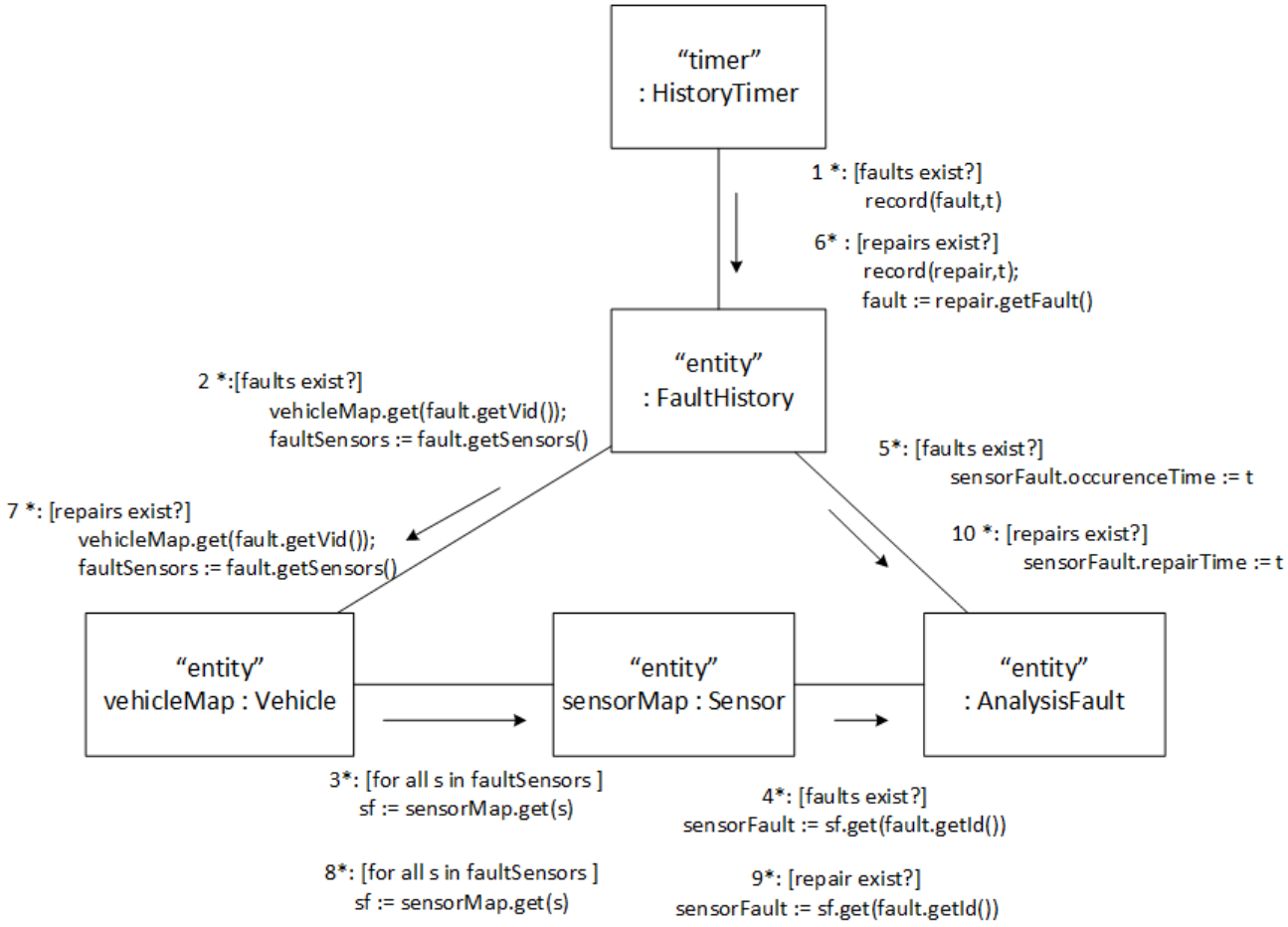

Figure 5.18: FaultHistory communication diagram.

Let $H^v_{\hat{F}}$ denote `FaultHistory`.vehicleMap and $H^v_{\hat{F}}(\hat{s})$ denote the set of faults that occurred on `Vehicle` v where the sensor instance of class $\hat{s}$ was fault involved. To populate and update the additional `HashMap` in response to incoming repairs, $\forall r \in R$, let v be the repaired `Vehicle`, and let f denote the `FaultDescription` repaired by r. Iterate $\forall \hat{s} \in f$.affectedSensors, where $\hat{s}$ is a fault-involved `Sensor` class that has been repaired by r, and note the repair time of the fault (f'), described by f, in the `FaultHistory`; that is, f'.repairTime $= t$, where $f' \in H^v_{\hat{F}}(\hat{s})$ and f'.faultDescription $= f$.

A similar process is done when a fault occurs. The `AnalysisFault`.occurrenceTime attribute is updated when a fault appears instead of the repair time. Iterate $\forall e \in F$, and let v be the `Vehicle` that has a fault, and let f denote the `FaultDescription` of e. Iterate $\forall \hat{s} \in f$.affectedSensors, where $\hat{s}$ is a fault-involved `Sensor` class of f, and note the occurrence time of the fault (f'), described by f, in the `FaultHistory`; that is, f'.occurrenceTime $= t$, where $f' \in H^v_{\hat{F}}(\hat{s})$ and f'.faultDescription $= f$.

After iterating all the input entity streams, the histories have recorded all events in the simulation for time t. Once the simulation is complete, this module will be notified that no further events need to be recorded, and finally it writes H_F, H_R, and H_S to its entity output stream.

Checking if a Fault Occurred in a Time Window To determine if any fault occurred during a target time period $T = [t_1, t_2]$, where t_1 and t_2 are a simulated day/time-unit and $t_1 \leq t_2$, given a sensor instance s from `Vehicle` v of `Sensor` class $\hat{s}$, it is sufficient to check whether s was fault-involved during T, since by definition no two active faults on v can share a fault-involved sensor. A fault occurred during T involving s if the following holds: $\forall f' \in H^v_{\hat{F}}(\hat{s})$, where f' is an `AnalysisFault` object, there is at least one f' such that the active fault period occurred during T, which can be expressed as $P \cap Q \neq \emptyset$, where P denotes the set of integers representing the days/time-units that fall within the time period when f' was active, such that $P \subseteq \mathbb{Z}, \forall i \in P$, f'.occurrenceTime $\leq i \leq f$.repairTime. Conversely, Q denotes the set of days found in the period T, which can be expressed as $Q \subseteq \mathbb{Z}, \forall i \in Q, t_1 \leq i \leq t_2$.

To simplify these notations, let all of the above be denoted as $\pi^v_{(t_1,t_2)}(\hat{s})$, where $\pi^v_{(t_1,t_2)}(\hat{s})$ $= 1$ if the sensor instance s was fault-involved during one of the days from t_1 to t_2, and $\pi^v_{(t_1,t_2)}(\hat{s}) = 0$ if s was not fault-involved during those days. This simplified notation will be used in later sections where the output of this module is used.

5.3.7 Analysis Phase

Problem Statement The analysis phase receives all the output from the previous simulation phases, analyzes it, and evaluates the performance of COSMO and ICOSMO. At this point, since ICOSMO has not yet been run in the simulation, it must therefore be run using all the output of the previous phases before the evaluation of both ICOSMO and COSMO can be made. Using the z-scores produced from the generation phase, they will be analyzed and the number of true positive (TP), true negative (TN), false negative (FN), and false positive (FP) rates will be calculated. To measure the performance of the anomaly detection algorithms, the area under ROC curves (AUC) will be used. Therefore, to produce a ROC curve, ROC curve points must be created, which consist of the true positive rate (TPR) and the false positive rate (FPR) for some COSMO deviation threshold between 0 and 1. To create multiple points, multiple thresholds must be used when computing the TPR and FPR. To choose the thresholds properly, all the z-scores obtained from the simulation are used as the thresholds. It is straightforward to compute the ROC curve points for the COSMO analysis. For each threshold, compute the FP, TP, FN, and TN rates. However, it is not as straightforward for ICOSMO, since ICOSMO's behavior changes depending on the threshold used; that is, the z-score output by ICOSMO will change depending on the COSMO threshold. Therefore, to analyze ICOSMO's performance, the simulation will have to be run for each threshold.

This phase can be broken down into four modules, namely, COSMO analysis, ICOSMO implementation, ICOSMO analysis, and the output module.

Input `SensorStatusEventHistory` entity, H_S, output by the History module (discussed in the history part of section 5.3.6.4), contains the full history of all the `SensorStatusEvents` output by the COSMO module of the simulation (discussed in the COSMO deviation detection part of section 5.3.6.3). It has the timeline of all the z-scores and other attributes of sensor instances and also is used to replay the `SensorStatusEvents` as a sensor status entity input stream for ICOSMO.

`FaultHistory` entity, H_F and $H_{\hat{F}}$, is another input from the History module, which contains a timeline of all the faults (and repair dates) output by the fault generation module (discussed in Fault generation model part of section 5.3.6.1). It is used to query for fault occurrence dates. `RepairHistory` entity, H_R, is another input from the History module, which contains the timeline of repairs that occurred in the simulation. It is used to determine when repairs occurred and also used to replay the repairs in the history as a repair entity input stream for ICOSMO.

Output A ROC curve is rendered and saved as an image file, the area under the curve for each algorithm (ICOSMO and COSMO) is also saved as a CSV file, and the history of the simulation is serialized and output as an object file. See Appendix B.5.4.1 for ROC curve point output class.

Parameters Threshold decimal precision parameter, p, indicating how many decimals should the thresholds have. In other words, a parameter used to skip a fraction of the thresholds that are similar. p is the number of decimal points to consider when treating z-scores as thresholds. All thresholds with more decimal points will be truncated.

Methodology

Simulation Logic H_S is used to compute the list of unique z-scores found in the entire simulation for all the `Algorithms`. To do so, all the `SensorStatusEvents` are gathered from the history, for all timestamps t and for each `Algorithm` a, which act as a partition key in H_S, $U \leftarrow U \cup e.\text{zscore}, \forall e \in H_S^a(t), \forall a, t \in H_S$, where U is the set of unique z-scores and e is a `SensorStatusEvent` object.

When simulations get large (for example, simulations with many vehicles, many sensors, high decimal precision for thresholds), there are many unique z-scores/thresholds. It becomes quite long to run a full ICOSMO simulation for each threshold. Therefore, I filtered out some z-scores using p. To get an idea of what precision to use, which investigates speed at the cost of accuracy, I ran experiments and they are discussed in section 5.6.5.1.

To manipulate the number of decimals for the threshold, the `java.math.BigDecimal` class is used, and the Java enumeration type `java.math.RoundingMode.HALF_UP` is used to specify that rounding up is required. See Appendix B.5.4.2 for a Java pseudo code example.

In this case, $precision = p$. So for example, with p=4, the original list of z-scores,

[0.1234,0.123456,0.123,0.123412,0.12]

would become

[0.1234,0.1235,0.1230,0.1234,0.1200]

. The set is then trimmed by removing duplicates, since the goal is to calculate the TPR and FPR for all unique thresholds. So, the resulting sorted threshold list would become

[0.1200,0.1230,0.1234,0.1235]

5.3.7.1 COSMO Analysis

Problem Statement The z-scores from the COSMO deviation detection module, which are found in the `SensorStatusHistory`, are used to compute the FPR an TPR for all thresholds. The thresholds are the unique list of z-scores which are computed at the beginning of the analysis phase. The points of a ROC curve are output for each `Algorithm`.

Input Unique set of thresholds, U. The `SensorStatusHistory` H_S and the `FaultHistory` H_F.

Output `ROCCurvePoint` entity output stream O, which primarily contains the TPR, FPR, and threshold as well as the algorithm id of the COSMO algorithm implementation (different histogram distances have different algorithm ids).

Parameters Two time windows: t_r, which is specified to decide how far in the future to check for faults, and t_l, which specifies how far in the past to look for fault occurrences in `FaultHistory` H_F.

Methodology

Simulation Logic Make sure to sort U, so the thresholds used are in increasing order. To count the TP, FP, FN, and TN rates, the following steps are performed: a) for each `Algorithm` $a \in H_S$, iterate all the thresholds $u \in$unique(U) which were truncated to a desired precision specified by p, b) for each the simulated time units $t \in H_S^a$, iterate each `SensorStatusEvent` object $e \in H_S^a(t)$ that occurred for time t, only considering e for COSMO sensors (e.isCOSMOSensor = true) (that is, only COSMO sensors are considered, since only selected sensors are used to detect faults), and c) let v denote the `Vehicle` and $\hat{s}$ denote the `Sensor` class of the sensor instance s that was used to produce e. Check whether s was fault-involved during the target period. If $\pi^v_{(t-t_l,t+t_r)}(\hat{s}) = 1$ (this notation is explained in the history part of section 5.3.6.4), s was fault-involved during the target time period from day $t - t_l$ to day $t + t_r$, and if it was equal to 0, s was not fault-involved during that time. Now that the presence of a fault is known, s is checked if it is deviating. That is, s is deviating if e.zscore $< u$, and is not deviating otherwise. A flag d is defined for notation simplicity, where $d = 0$ when e.zscore $< u$ and $d = 1$ when e.zscore $\geq u$. The mathematical expressions below summarize how the TP, FN, FP, and FN rates are computed:

$$\text{TP} = \sum_{a \in H_S} \sum_{u \in U} \sum_{t \in H_S^a} \sum_{e \in H_S^a(t)} \left[d \cdot \pi^v_{(t-t_l,t+t_r)}(\hat{s}) \right]$$

$$\text{FN} = \sum_{a \in H_S} \sum_{u \in U} \sum_{t \in H_S^a} \sum_{e \in H_S^a(t)} \left[(1-d) \cdot \pi^v_{(t-t_l, t+t_r)}(\hat{s}) \right]$$
$$\text{FP} = \sum_{a \in H_S} \sum_{u \in U} \sum_{t \in H_S^a} \sum_{e \in H_S^a(t)} \left[d \cdot [1 - \pi^v_{(t-t_l, t+t_r)}(\hat{s})] \right]$$
$$\text{TN} = \sum_{a \in H_S} \sum_{u \in U} \sum_{t \in H_S^a} \sum_{e \in H_S^a(t)} \left[(1-d) \cdot [1 - \pi^v_{(t-t_l, t+t_r)}(\hat{s})] \right]$$

Appendix B.5.5.1 shows the Java pseudo code that summarizes the logic for counting TP, FP, FN, and TN rates.

Once the rates are computed, the TPR and FPR are computed as well as the other metrics. With these metrics, a `ROCCurvePoint` of the form $\{a,TPR,FPR,u,...\}$ is created and output to O.

5.3.7.2 ICOSMO Implementation

Problem Statement ICOSMO is implemented to enhance the accuracy of COSMO. ICOSMO receives the z-scores produced by the COSMO approach's output, which is discussed in the COSMO deviation detection part of section 5.3.6.3. As repairs and `SensorStatusEvent` objects are input to ICOSMO, it updates the attributes of its internal map's sensor instances. As repairs occur, ICOSMO will adjust fault-involved sensors' sensor potential contribution (SPC) and sensor contribution (SC), which are used to remove or add sensors from the COSMO sensor selection. Since sensors are dynamically selected in ICOSMO, a timestamp must be attached to all the `SensorStatusEvent` objects output, since a sensor may be selected or un-selected at different points in time. To be able to count the TP, FP, FN, and TN rates, the time when a sensor was selected or un-selected is necessary.

Input Repair entity input stream R. `SensorStatusEvent` entity input stream S. Time t.

Output Timestamped `SensorStatusEvent` entity output stream O, which contains timestamped `SensorStatusEvent` objects. The format of an entity $o \in O$ is as follows: $o = \{t, e\}$, where t is time that the `SensorStatusEvent` object e was output.

Parameters COSMO deviation threshold d. The number of days to look in the past, t_l, when looking for deviations when repairs occur. Simulated information retrieval enumeration type i, which is either `RECALL` or `PRECISION`. i is used to decide if desired recall d_r or precision d_p will be met when estimating fault-involved sensors using the simulated information retrieval algorithm (IRA). Number of fault-involved sensors to estimate n. $\Delta^+_{\texttt{SC}}$ is the sensor contribution increase modifier, $\Delta^-_{\texttt{SC}}$ is the sensor contribution decrease modifier, $\Delta^+_{\texttt{SPC}}$ is the sensor potential contribution increase modifier, and $\Delta^-_{\texttt{SPC}}$ is the sensor potential contribution decrease modifier. $D_{\texttt{SC}}$ is the default sensor contribution, and $D_{\texttt{SPC}}$ is the default sensor potential contribution. θ is the staleness threshold for the mean SC, and ω is the candidacy threshold for the mean SPC. $maxRm$ is the maximum number of stale selected sensors (COSMO sensors) to remove from the COSMO sensors when dynamically adjusting the selected sensors. $maxAdd$ is the maximum number of candidate sensors that can be added to the COSMO sensors.

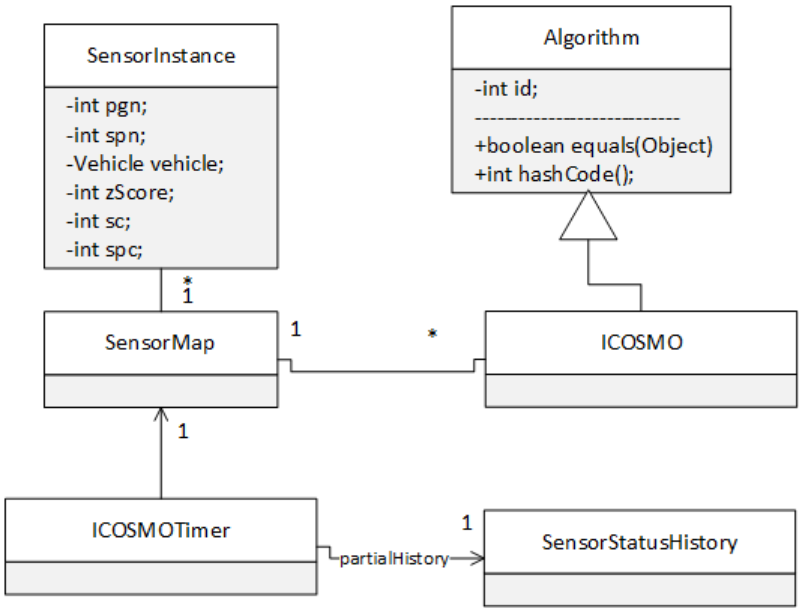

Figure 5.19: ICOSMO Implementation UML Class Diagram.

Methodology Figure 5.19 illustrates the simple class diagram of the classes used in this module.

Configuration A `SensorMap` M is initialized in the same way the `SensorMap` is configured in the COSMO deviation detection module discussed in section 5.3.6.3. A partial `SensorStatusHistory` $\hat{H}_S$ is also used to keep track of the history of z-scores in the temporary ICOSMO simulations. $\hat{H}_S$ is used by ICOSMO for looking up z-scores for deviations in the past when repairs occur. There is also a `java.util.HashMap` object, N, of the form `HashMap<Algorithm,Integer>`, which is used to keep track of the number of removed and added COSMO sensors, where $N_a = x$ denotes the application of ICOSMO on `Algorithm` a; when $x < 0$, after addition and/or removal there is a net loss of number of sensors selected, and when $x > 0$ there is a net gain of number of selected sensors. When ICOSMO is initialized, all unnecessary object references in M, $\hat{H}_S$, and N are cleared as follows: reset all the attributes of sensor instances found in M to default values; Remove all the `SensorStatusEvent` objects from the internal lists in $\hat{H}_S$, by a) iterating all the `Algorithm` partition keys $a \in \hat{H}_S$, b) iterating all time entries $t \in \hat{H}_S^a$, c) removing all references to `SensorStatusEvents` in $\hat{H}_S^a(t)$ using the `java.util.List.clear()` function, which sets all elements in the list to null, and d) assigning for all `Algorithms` $a \in N, N_a = 0$. This avoids unnecessary repeated garbage collection and memory allocation of data structures that can be reused over multiple ICOSMO executions.

Simulation Logic

Processing and Time-stamping z-scores

For a new simulation day time t, iterate all the `Algorithm` partition keys $a \in \hat{H}_S$. Iterate all the `SensorStatusEvent` objects e of `Algorithm` a from the input S. Let v denote the `Vehicle` of e, $\hat{s}$ the `Sensor` class of e, and $\hat{z}$ the z-value of e. Where the $\hat{z}$

is result of the Most Central Pattern without taking the moving window average. The moving average is tracked in this module, so only the z-values are required to rebuild the z-score moving average. Retrieve the `SensorInstance` object reference s from M, where $s = M$.getSensorInstance(a,v,$\hat{s}$). Add $\hat{z}$ to the moving average z-score queue of s. Compute z, the z-score of s now that $\hat{z}$ has been added, which is the mean of all elements in the queue. Now that s has been updated, output e with a timestamp by doing the following: create a new `SensorStatusEvent` object $\hat{e}$, such that $\hat{e} = e$. Make sure to reflect the current status of s in $\hat{e}$, since the z-score or COSMO sensor selection status of s may be different from e, so $\hat{e}$.zscore = s.zscore and $\hat{e}$.isCosmoSensor = s.isCosmoSensor. Output $\hat{e}$ with a timestamp, as follows: $O \leftarrow O \cup \{t, \hat{e}\}$.

Limiting the Number of Sensor Selection Changes

When the sensor selection is dynamically adjusted by ICOSMO, the parameters $maxAdd$ and $maxRm$ are used to limit the sensor selection changes. The main purpose of this implementation choice is to make sure that there is at least one selected sensor, since having zero selected sensors would not make sense. Whenever ICOSMO adjusts the sensors selected, it keeps track of the number of sensor selection changes it has made as follows: when a `Sensor` is added to the COSMO sensors for `Algorithm` a, increment N_a by 1, and when a `Sensor` is removed from the COSMO sensors, decrease N_a by 1. To limit the sensor selection scope, a candidate sensor for `Algorithm` a can be added to the COSMO sensor if and only if $N_a < maxAdd$, and a stale sensor can only be removed from the COSMO sensors if and only if $N_a > -(maxRm)$. See algorithm 4 for the pseudo code that summarizes the logic used to dynamically adjust the selected sensors.

Algorithm 4 Limiting the dynamic sensor selection.

```
1: procedure ADDCANDIDATESENSOR(a,ŝ)   ▷ a algorithm,ŝ non-cosmo sensor class to add
2:     if N_a < maxAdd then
3:         sensorSelectionHelper(a,ŝ,true)
4:         N_a = N_a + 1
5: procedure REMOVESTALESENSOR(a,ŝ) ▷ a algorithm,ŝ cosmo sensor class to remove
6:     if N_a > −maxRm then
7:         sensorSelectionHelper(a,ŝ,false)
8:         N_a = N_a − 1
9: procedure SENSORSELECTIONHELPER(a,ŝ,f)   ▷ ŝ sensor class to change select, f true when adding COSMO sensor, false otherwise
10:    S = M.getSensorInstances(a,ŝ)
11:    for s ∈ S do
12:        s.spc =D_spc
13:        s.sc =D_sc
14:        s.cosmoSensorFlag = f
15:        s.zScoreWindow.clear()
```

Adjusting Sensor Rankings

The logic for updating SPC and SC, which will potentially remove or add COSMO sensors from the selected sensors, is as follows. Iterate all the repairs $r \in R$. Let f denote the `FaultDescription` object that was repaired by r. Let P denote the actual fault-involved sensors of f and $\overline{P}$ denote the actual sensors that are not fault-involved with f (see section 5.3.6.1 for how P and $\overline{P}$ are defined). Estimate the set of fault-involved sensors X, X =`IRA_RECALL(`P`,`$\overline{P}$`,`d_r`,`n`)` when i =`RECALL` and X =`IRA_PRECISION(`P`,`$\overline{P}$`,`d_p`,`n`)` when i =`PRECISION`. Run the `onRepairInsert(`r`,`X`)` procedure described in algorithm 5.

5.3.7.3 ICOSMO Analysis

Problem Statement To evaluate the performance of ICOSMO, most of its simulation must entirely be replayed. The sensor data generation is the only portion of the simulation that is not repeated, since ICOSMO only requires the z-scores of the sensor instances. The approach used for analyzing the COSMO approach's performance, which is discussed in the COSMO analysis part of section 5.3.7.1, cannot be applied here for counting the TP, FP, FN, and TN rates, since the z-scores will change and are not static for ICOSMO. Therefore, for each threshold the entire simulation must be re-run to obtain a single ROC curve point for ICOSMO. This is the case, as the z-scores output by ICOSMO changes depending on the COSMO threshold used, since it may exclude or include COSMO sensors, which will affect the z-scores. The `SensorStatusHistory` and `RepairHistory` are replayed to ICOSMO, and ICOSMO outputs its `SensorStatusEvents` for all time units of the simulation. Each output set produced by replaying the simulation for ICOSMO will be used with the `FaultHistory` to count the TP, FP, FN, and TN rates, to produce `ROCCurvePoints` as output.

Input `FaultHistory` H_F, `SensorStatusHistory` H_S, `RepairHistory` H_R from the history module, which is discussed in the history part of section 5.3.6.4. A set of unique thresholds U to use for deviation thresholds.

Output `ROCCurvePoint` entity output stream O, which primarily contains the TPR, FPR, and a COSMO deviation threshold used to compute the TPR and FPR, as well as the algorithm id of the ICOSMO algorithm implementation (different histogram distance have different algorithm ids).

Methodology The history replay module has an `ICOSMO` object that is initialized each time the simulation is re-run for ICOSMO. All internal object associations in the `ICOSMO` instance are reset to get ready for the new simulation run. Since simulations can become large and require a lot of memory, instead of creating a new `ICOSMO` instance, the internal `ICOSMO` object associations (references) are set to `null` to let the Java garbage collection take care of it. Otherwise the Java Virtual Machine will be spending most of its time doing garbage collection in the case when $|U|$ is large.

Simulation Logic Once U has been populated, the histories are iterated in chronological order and the TP, FP, FN, and TN rates, are counted. For analyzing ICOSMO, H_S must be replayed for each $u \in U$, where the threshold of ICOSMO is set to u, and from a chronological order the history is replayed for ICOSMO. Iterate all time $t \in H_S$ and create a Repair entity input stream R' such that $R' = H_R^v(t), \forall v \in H_R$. Provide R'

Algorithm 5 Adjusting the sensors selected.

1: **procedure** onRepairInsert(a,t,v,P,$\overline{P}$,X) ▷ time t, algorithm a,vehicle v, X is the estimated fault-involved sensor set, P actual fault-involved sensors, $\overline{P}$ actual non-fault-involved sensors
2: f = `false`
3: **for** $\hat{s} \in X$ **do** ▷ at least one deviation occurred?
4: $s = M$.getSensorInstance(a,v,$\hat{s}$)
5: **if** s.cosmoSensorFlag = `true` **then**
6: **if** $\pi^{v}_{(t-t_l,t)}(\hat{s}) = 1$ **then**
7: f = `true`
8: **for** $\hat{s} \in X$ **do**
9: d = `false`
10: $s = M$.getSensorInstance(a,v,$\hat{s}$)
11: **if** s.cosmoSensorFlag = `true` **then**
12: **if** $\pi^{v}_{(t-t_l,t)}(\hat{s}) = 1$ **then**
13: d = `true`
14: **if** s.cosmoSensorFlag = `true` **then** ▷ COSMO sensor?
15: **if** d = `true` **then** ▷ deviation occurred for s?
16: s.sc = s.sc + $\Delta^{+}_{\texttt{SC}}$
17: **else** ▷ s fail to deviate in response to fault
18: s.sc = s.sc - $\Delta^{-}_{\texttt{SC}}$
19: **if** isStale(a,$\hat{s}$) = true **then**
20: removeStaleSensor(a,$\hat{s}$)
21: **else** ▷ non-cosmo sensor
22: **if** f = `true` **then** ▷ deviation occurred for some $s \in X$?
23: s.spc = s.spc - $\Delta^{-}_{\texttt{SPC}}$
24: **else** ▷ no deviation linked to fault
25: s.spc = s.spc + $\Delta^{+}_{\texttt{SPC}}$
26: **if** isCandidate(a,$\hat{s}$) = true **then**
27: addCandidateSensor(a,$\hat{s}$)
28: **procedure** isCandidate(a,$\hat{s}$) ▷
29: $S = M$.getSensorInstances(a,$\hat{s}$)
30: $\mu = 0$
31: **for** $\hat{s} \in S$ **do**
32: $\mu = \mu + s$.spc
33: $\mu = \frac{\mu}{|S|}$
34: **return** $\mu \geq \theta$
35: **procedure** isStale(a,$\hat{s}$) ▷
36: $S = M$.getSensorInstances(a,$\hat{s}$)
37: $\mu = 0$
38: **for** $\hat{s} \in S$ **do**
39: $\mu = \mu + s$.sc
40: $\mu = \frac{\mu}{|S|}$
41: **return** $\mu \leq \sigma$

as the input to ICOSMO. It is worth noting the methodology of implementing ICOSMO in this simulation is discussed in the ICOSMO implementation part of section 5.3.7.2. The `SensorStatusEvent` object for time t is also provided to ICOSMO's input. Let S' denote the `SensorStatusEvent` entity input stream that will be provided to ICOSMO, $S' = H_S^a(t), \forall a \in H_S$. ICOSMO then outputs a set of timestamped `SensorStatusEvents` O', where O' is then analyzed to count the TP, FP, FN, and TN rates produced by ICOSMO for threshold u, such that only the COSMO sensors' z-scores are considered, since only they can be used to detect faults. The temporary result for each u is output to O, as $\{\text{TPR,FPR,}u,\hat{a}\}$, where $\hat{a}$ is the ICOSMO version of the `Algorithm` a. In other words, $\hat{a}$ indicates the output is the ICOSMO `Algorithm` applied to the z-scores produced by `Algorithm` a. More formally, evaluating ICOSMO consists of applying the pseudo code logic found in algorithm 6.

Algorithm 6 Evaluation of ICOSMO's performance.

procedure EVAL_ICOSMO(H_S,H_F,H_R,U)
  $O = \{\}$
  for $a \in H_S$ **do** ▷ iterate all algorithms
    for $u \in U$ **do** ▷ iterate all thresholds
      $TP = 0$
      $FN = 0$
      $FP = 0$
      $TN = 0$
      ICOSMO.reset()
      for $t \in H_S^a$ **do** ▷ replay sensor status history
        $R' = H_R^v(t), \forall v \in H_R$
        $\hat{O} =$ `run_ICOSMO(`$t, H_S^a(t), R'$`)`
        for $\{\hat{t}, \hat{e}\} \in \hat{O}$ **do** ▷ iterate all timestamped icosmo SensorStatusEvents
          if $\hat{e}$.isCosmoSensor **then** skip non-cosmo sensor zscores
            $v = \hat{e}$.vehicle
            $s = \hat{e}$.sensor
            $d = \hat{e}$.zscore $\geq u$
            $\sigma = \pi^v_{(\hat{t}-t_l,\hat{t}+t_r)}(s)$
            $TP = TP + d\sigma$
            $FN = FN + (1-d)\sigma$
            $FP = FP + d(1-\sigma)$
            $TN = TN + (1-d)(1-\sigma)$
      $TPR = \text{computeTPR}(TP,FP,TN,FN)$
      $FPR = \text{computeFPR}(TP,FP,TN,FN)$
      $a' = a$.toICOSMO()
      $O = O \cup \{a', u, TPR, FPR, ...\}$ ▷ output roc curve point
  return O

ICOSMO does not have the `FaultHistory` replayed, since, by definition, ICOSMO is not aware of when faults occurred. The assumptions state that a vehicle service record

database (VSRDB) is available, and therefore ICOSMO is only aware of when repairs occur.

5.3.7.4 Simulation Output

Problem Statement This module outputs all the relevant data produced by the simulation, including the history of the simulation, ROC curve points as the evaluation metric, the area under the curve of the ROC curves, and a rendered image of the ROC curves.

Input ROC curve point entity input stream P, is the input, which is the output of the analysis phase discussed in section 5.3.7. The History of the simulation, which is output by the history module (discussed in the history part of section 5.3.6.4), is input to this module, which includes the `FaultHistory` H_F, the `SensorStatusHistory` H_S, and the `RepairHistory` H_R.

Output CSV file of all the ROC curve points $f_{\texttt{CSV}}$ for all the algorithms, CSV file of the area under the curve (AUC) of each algorithm's ROC curve chart $f_{\texttt{AUC}}$, the ROC curve rendered image output file $f_{\texttt{ROC}}$, the area under the curve (AUC) bar chart $f_{\texttt{BC}}$, the log file of the simulation $f_{\texttt{LOG}}$, and the output serialized history file $f_{\texttt{H}}$, which contains H_S, H_F, and H_R.

Parameters Some parameters to this module include the output file paths of the output files and also the R script file path to be executed to analyze $f_{\texttt{CSV}}$.

Methodology

Output in Java Language

With the help of the `java.nio` package, using `java.nio.file.Path`, `java.nio.file.Paths`, `java.nio.file.Files`, and `java.nio.file.StandardOpenOption`, create the output files. First copy the simulation log to the file $f_{\texttt{LOG}}$. Then create $f_{\texttt{CSV}}$, initially an empty file, and append the following CSV header to it:

```
experiment id,algorithm_id,algorith_name,tp,tn,fp,fn,tp rate,fp
    rate,threshold,accuracy,fscore
```

Iterate all the ROC points created in the performance analysis modules (see ICOSMO and COSMO analysis parts in section 5.3.7.3 and 5.3.7.1, respectively), which are found in P. $\forall p \in P$, create a `String` object s and append all attributes of p to s, such that each attribute is separated by a comma. At the end of s, append a new line.

The history of the simulation is then saved for future use. To output the history, Java object serialization is used to save the histories to a file. That is, the entire object structures of the H_F, H_S, and H_R are serialized and saved to a file by wrapping them in a `HistoryEvent` object and saving them to the file $f_{\texttt{H}}$, using the method illustrated in Appendix B.5.6.1.

Objects saved this way must implement the `java.io.Serializable` interface. With the help of the `java.io` package classes, the histories are saved to a file and can be

read to replay simulations with different parameters. The simulations have a lot of non-deterministic behavior, because they use random number generation, which means running two simulations with identical input and parameters will most likely produce different results. By saving histories of a simulation, the non-determinism of the synthetic data generation is removed, since the z-scores produced by the COSMO approach remain the same. Note that there is still non-determinism in ICOSMO, since it also uses random numbers in its simulated IRA, but at least some experimental control is added by removing the non-determinism of generating faults, repairs, and synthetic data. This provides the flexibility to re-run simulations on the same results of past simulations, but varying key parameters and observing the effects they have on the results

Output in R Language Once all the output files from the Java portion of the output logic, $f_{\texttt{CSV}}$ is used by the R programming language to render a ROC curve chart in the file $f_{\texttt{ROC}}$, and to create the CSV $f_{\texttt{AUC}}$ with the AUC for each ROC curve. The R programming language version used was R 3.6.0 on a Windows 7 machine. Details of the R implementation can be found in section B.6.

5.4 STO Sensor Data Sampling Analysis

In the fleet simulations (note that its implementation is discussed in section 5.3), data from the primary STO data dump was sampled (see the J1939 Sensor Data part in section 5.3.5.1). Not all the sensor data was included in the simulation for efficiency reasons. I wanted to know how much sensor data I could filter before hindering the sample mean significantly. To investigate this, I analyzed the sample mean confidence levels of the sensor data. I wanted to know how likely the sample mean, the mean of the set of sampled sensor readings, would fall within an interval of the actual mean. Section 2.12.3 goes into the details of the mathematics required to perform this analysis.

In my early simulation experiments, the simulations that sampled 10% of the readings were quite fast. Therefore, I decided to analyze the confidence intervals when 5%, 10%, and 15% of sensor readings were sampled. For each of these cases, I wanted to know the estimated probability that the sample mean would fall within 5%, 10%, and 15% of the actual mean. To analyze this, the actual mean and standard deviation of each sensor stream were required, which I found by analyzing the STO primary data dump (which is discussed in the data exploration part of section 5.2.4). Using the actual mean μ and standard deviation σ of each sensor, I computed the confidence levels for the following three cases:

- $P(\mu_{-0.05} < \overline{X} < \mu_{0.05})$
- $P(\mu_{-0.1} < \overline{X} < \mu_{0.1})$
- $P(\mu_{-0.15} < \overline{X} < \mu_{0.15})$

where $\mu_i = i\mu + \mu$, in other words, the estimated probability that the sample mean $\overline{X}$ of each sensor stream falls within 5%, 10%, and 15% of the actual mean. I computed the

Case	i	f_p	mean	std	min	max
1	0.05	0.95	0.98080	0.06098	0.65442	1.000
2	0.05	0.90	0.99040	0.03633	0.81775	1.000
3	0.05	0.85	0.99686	0.01517	0.89767	1.000
4	0.10	0.95	0.99837	0.00840	0.94076	1.000
5	0.10	0.90	0.99971	0.00137	0.99236	1.000
6	0.10	0.85	0.99998	0.00014	0.99891	1.000
7	0.15	0.95	1.00000	0.00000	1.00000	1.000
8	0.15	0.90	1.00000	0.00001	0.99994	1.000
9	0.15	0.85	0.99990	0.00060	0.99534	1.000

Table 5.5: The mean estimated probabilities, for 61 sensors, that their sample mean fall within confidence intervals, where i denotes the confidence interval as follows: $P(\mu_{-i} < \overline{X} < \mu_i)$ and f_p denotes the fraction of sensor readings filtered (1 - f_p denotes the fraction of readings sampled).

confidence levels for each of the 61 sensors that were chosen (see sensor selection part of section 5.2.4.1 for details on the sensor selection strategy). Table 5.5 illustrates the results of the confidence interval analysis.

Table 5.5 reveals that the majority sampling strategies are very likely to produce a sample mean with small error (a short distance from the actual mean), on average. That is, for each case their confidence level is above 98%. There are three rows in the table that are interesting.

- The 4th row, where $i = 0.10$ and $f_p = 0.95$, has a minimum confidence level of 0.941, while the other cases for $i = 0.10$ have a minimum confidence level above 0.99.
- For the rows such with $i = 0.05$, the minimum confidence level is relatively low. However, this is only for a few sensors, since the mean confidence level are all above 98%. Although the minimum confidence levels raise a bit of concern, I argue it will not have much effect on the outcome of the simulations, since the simulated vehicles will be adding their own white noise to each sensor reading and only a few sensors will have a sample mean that will likely be more than 5% away from their actual mean.

Figure A.11 illustrates in more detail the confidence levels of each sensor stream for each sampling strategy.

5.5 Synthetic Sensor Data Generation Analysis

5.5.1 Sensor Behavior Parameter Analysis

This section analyzes the effects the parameters from equation 5.1 have on the original dataset by comparing the original dataset D to the synthetic dataset produced. The

examined parameters are explained in the sensor data generation model part of section 5.3.6.2. To begin the analysis, three datasets were generated, with 1000 samples each, namely, D, N, and . The original/base dataset created, D, was normally distributed with mean = 0 and standard deviation = 1. The base/injected noise dataset created, N, was uniformly distributed between -7 and 7. The additive white noise dataset, ω, shared the same distribution as D. The manipulated parameters in equation 5.1 are the amplification factor, α, and the probability of sampling from N, p_x (let p denote p_x in this section). I ran nine data generation experiments with the following parameter configuration: ($\alpha = 2$, $p = 0$), ($\alpha = 2$, $p = 0.4$), ($\alpha = 2$, $p = 0.7$), ($\alpha = 1$, $p = 0$), ($\alpha = 1$, $p = 0.4$), ($\alpha = 1$, $p = 0.7$), ($\alpha = 0.5$, $p = 0$), ($\alpha = 0.5$, $p = 0.4$), and ($\alpha = 0.5$, $p = 0.7$), each creating a synthetic dataset of 1000 samples. The results of the analysis are illustrated in figure A.10.

The results reveal that smaller values of α produce synthetic datasets that squeeze towards D's mean, and conversely larger α values stretch the original dataset D. Furthermore, as p increases the synthetic dataset becomes proportionally similar to the injected noise dataset N. Other conducted experiments revealed that the parameter β affects N in a similar way as α affects D.

5.5.2 Stability Analysis

In this analysis, the effects of the p_x values are observed (for more details on p_x see section 5.3.6.2) on the stability (for more information on stability, see section 2.15.4.4) when using the Hellinger distance. It is worth noting that [82] states an interesting signal has a low stability value and an uninteresting sensor signal has a high stability value. Similar experiments to those conducted in section 5.5.1 were conducted. Instead of generating a synthetic dataset D, a sensor stream from the primary STO dataset (see section 5.1.4) was used as the original dataset, which had 215348 samples. Figure A.9 illustrates the histogram of the sensor's distribution. There are two experiments that were run in this section, which both used D as the original dataset, but use different injected noise, N, and additive white noise, ω, synthetic datasets. In both experiments, multiple synthetic dataset pairs were generated using equation 5.1. Both datasets in these pairs were modeled using a histogram and compared by computing their stability (the stability is defined in equation 2.21), using the Hellinger distance. A dataset pair was generated for each pair of p_0 and p_1. Both p_0 and p_1 were incremented by steps of 0.025. In **Experiment 1**, N was uniformly distributed between -7.8125 and 4.5625. ω was normally distributed with mean = 0 and standard deviation = 3. In **Experiment 2**, N was uniformly distributed between -9.765625 and 3.421875. ω was uniformly distributed between -3 and 3.

Let Δp denote $p_1 - p_0$. Experimental results reveal that, in general, as Δp increases the stability also increases. These results were useful in the early phases of the simulation design, since they allowed the stability of sensors to be manipulated by adjusting the p_0 and p_1 parameters of the simulations. Experiment 1 reveals that with uniformly distributed additive white noise, the stability increases at a much faster rate as Δp increases, and the maximum stability reaches approximately 0.9, which is illustrated in figure A.7. Experiment 2 reveals that for normally distributed additive white noise, as Δp increases, the

Algorithm 7 Computing stability vs. injected noise on a STO sensor stream.

```
1: procedure GENERATEHISTOGRAM(D,N,ω,p)                                  ▷
2:     n = |D|                                          ▷ number of samples
3:     θ = {}                                           ▷ empty synthetic dataset
4:     for i = 0 to n do
5:         θ = θ ∪ apply equation 5.1 using (d_i,N_i,ω_i, and p)
       return createHistogram(θ)
6: procedure GENERATEALLPAIRS(D,N,ω)                                     ▷
7:     steps = {0, 0.025, 0.05, 0.075, ..., 0.975, 1.0}
8:     for p_1 ∈ steps do
9:         p_2 = p_1 + 0.025
10:        while p_2 < 1.0 do
11:            h_1 = generateHistogram(D,N,ω, p_1)
12:            h_2 = generateHistogram(D,N,ω, p_2)
13:            computeStability(h_1,h_2)
14:            ...
15:            p_2 = p_2 + 0.025
```

stability increases at a much slower rate (compared to experiment 1), and the maximum stability does not go beyond 0.6, which is illustrated in figure A.8.

5.6 Fleet Simulation Result Analysis

This section presents and analyzes the results produced by the fleet simulation, whose implementation and design are discussed in section 5.3. A fleet of a variable number of vehicles was simulated to evaluate the performance of the COSMO approach and ICOSMO approach using a variety of different histogram distance measures (these histogram distances can be found in equation 5.4). For notation purposes, in this section when I refer to a histogram distance, I am referring to when COSMO or ICOSMO use the histogram distance.

5.6.1 Goal

There are three things I analyzed, namely, the accuracy of the various histogram distance measures when used in both ICOSMO and COSMO, the effects of the ICOSMO parameters on ICOSMO's performance, and the performance differences between COSMO and ICOSMO; that is, the effects ICOSMO has on COSMO's results.

5.6.2 Methodology

To evaluate the performance of various instances of COSMO and ICOSMO, ROC curves are generated and the area under their curve (AUC) is analyzed. There are two types of

experiments, Type A and Type B, which I ran: **type A**, which re-uses a history produced by an older simulation. Type A simulations do not re-create a history; instead they only run a partial simulation (ICOSMO) on the z-scores and use the faults and repairs of the history to calculate TPRs and FPRs to produce a ROC curve. Experiments of **type B** create a new history by running an entire simulation, which produces sensor data streams, faults, repairs, and z-scores. For both types of experiments, multiple iterations/simulations are run using the same configuration to obtain average performance results.

5.6.3 Simulation Configuration

The simulations' behavior is defined by their configuration parameters. Each experiment manipulates the configuration parameters to observe the effects on the results. For all experiments, a model gathering period of 7 days (parameter m, defined in section 5.3.6.3) was used. The moving average window size for the z-score was 30 days (parameter w, defined in section 5.3.6.3). All the experiments included 61 sensors (parameter W, defined in section 5.3.5.1) from the STO primary dataset, which were chosen using methods discussed in section 5.2.4.1. Each experiment sampled 10% of the sensor readings from the primary STO data dump; that is, 90% of readings were dropped (parameter $f_p = 0.90$, defined in section 5.3.5.1). This was done to keep the sample mean reasonably close to the actual STO data dump's sensor means and to reduce the time required to perform multiple simulations (for more details on this sensor data sampling, see section 5.4). In general, the probability of occurrence for the various faults varied between 0.01% and 1% (in some cases single-sensor failure faults were defined to be much more rare, such as $1\text{x}10^{-7}$ for example). The repair probability for most faults varied between 1% and 40%, with a minimum days before repair that varied from 1 to 20 days.

The experimental results analyzed in this section will also list the ICOSMO parameters used to configure the simulation that output the results. For more details on the parameters of ICOSMO, see section 5.3.7.2. The initial values of ICOSMO's parameters are important, since they determine the initial behavior of ICOSMO and have a strong influence on its average behavior throughout its execution cycle. A poor choice of initial values will lead to poor ICOSMO behavior and a good choice will lead to more accurate behavior. However, I did not know which initial values to choose when I first began running simulations, so I started by randomly assigning values to ICOSMO's parameters. The first few sets of experiments with randomly assigned parameters produced no change in COSMO's behavior; that is, ICOSMO had no effect on the accuracy of COSMO. I kept changing ICOSMO's initial values and running experiments until ICOSMO produced different accuracies than COSMO. I used these initial values as a foundation for determining the initial values of the parameters used in future simulation experiments. It is worth noting that the initial values found, those that affect COSMO's accuracies, are tied to the primary STO dataset. It may be the case that for other datasets these initial values would be a poor choice (would have no effect on COSMO's accuracy). However, due to the limited amount of real data, I could not explore ICOSMO's initial values for additional datasets.

5.6.3.1 Fault Descriptions

In all the experiments, I defined 25 fault descriptions (for more details see section 5.3.6.1) that describe physical component failures, and I defined individual sensor failures (where a faulty sensor stops reporting accurate readings) for each sensor on the vehicles. These faults could occur on any vehicle and were defined by a set of sensor behaviors (see section 5.3.6.2 for more details on sensor behavior and table 5.4 for a description of all the attributes of a sensor behavior). Although the fault descriptions do not necessarily define the behavior of a real fault accurately, the purpose of the simulation is to affect the distribution of the sensor data streams to observe their effects on deviation detection.

An example of any type of overheating fault can be simulated by increasing the average white noise of a temperature sensor. More detailed examples of fault descriptions are found below. Note that to limit the scope of the result analysis, the entire list of fault descriptions from the simulations is not discussed in this thesis. The parameters mentioned in this section are from the data generation equation 5.1.

1. **Engine Overheating**: for each sensor behavior, the type is set to "MODIFY" to adjust the current sensor readings. All the sensors reporting various engine liquid temperatures are increased by a fraction. The engine cooling fan would also increase in speed to try to cool the engine. Suppose the units used for the cooling fan's speed is represented in percentage of utilization (0-100% utilized), so the readings of the engine fan sensor could be increased by 50%.

2. **Run-away Cooling Fan**: for each engine temperature sensor behavior, the type is set to "MODIFY" to adjust the current sensor readings. All the engine liquids would be much cooler, so the mean additive white noise to add would be a negative value. To represent the run-away fan, which is running at 100% capacity, the type of sensor behavior would be "NEW", since the original sensor behavior of the fan speed will be overwritten and set to always output 100% speed. The table 5.2 gives a complete example of this fault.

3. **Single Sensor Failure**: a faulty sensor's original behavior (the base STO dataset samples) is multiplied by 2, p_0 is increased by 10%, and p_1 is increased by 20% to be more likely to sample from the base noise dataset, and the standard deviation of the white noise is increased by 2. The following parameters are affected: $\alpha_{vs} = \alpha_{vs} + 1$, $p_0 = p_0 + 0.1$, $p_1 = p_1 + 0.2$, and $\sigma(\omega_{vs}) = \sigma(\omega_{vs}) + 2$, where $\sigma(\omega_{vs})$ is the standard deviation of the white noise.

5.6.3.2 Sensor Data Behavior

I wanted to ensure that each vehicle's sensors would be similar but not identical. The base noise amplification factor, the sensor reading amplification factor, and the additive white noise attributes are used to ensure that each vehicle sampling from the same primary STO dataset and the same base injected noise dataset would produce slightly different data

streams. Most sensor behavior's α and β attributes were random variables sampled from a normal distribution with mean = 1 and standard deviation = 0.05.

To define the white noise for each sensor's behavior, I analyzed the statistics gathered from the STO data dump. Using these sensor statistics, I defined the white noise to try to ensure the statistics of the original distribution remained unchanged. For example, for a STO sensor with mean 100 and standard deviation of 10, its white noise with mean = 0 and standard deviation between 1 and 5 would be appropriate. This way, the added noise should distinguish each vehicle's sensor output stream but generally keep their distributions similar to the original distribution. A bad white noise assignment, for example, would be setting the mean = 50 and standard deviation = 100 of the white noise. This would completely change the original sensor's distribution.

5.6.4 Histogram Distance Analysis

This section analyzes the performance of the various histogram distances when used in COSMO and ICOSMO. Since Type A experiments do not re-create z-scores, COSMO's performance results remain constant for all iterations of these types of experiments. Therefore, Type B experiments are required to produce different COSMO performance results for each iteration, which will enable the average performance evaluation of the histogram distances.

5.6.4.1 Small Simulation - Experimental Results

This Type B experiment set is configured as follows: it has 60-day simulations, high and low sensor white noise, 8 vehicles defined, 64 iterations, very rare single-sensor faults (0.0000001% to 0.000001% occurrence rates, 10% to 20% repair occurrence rates, and a minimum of 2 days before repaired), and a variety of selected sensors from 2 to 32.

Experimental results reveal that the histogram distances that produce the worst accuracy, on average, are the Cosine, Fidelity, and Intersection distances. The other histogram distances perform better than these and are similar to each other. This is illustrated in figure 5.20.

5.6.4.2 Large Simulation - Experimental Results

These Type B experiment sets are configured as follows: (48 vehicles, 250-day), (32 vehicles, 250-day), and (20 vehicles, 180-day) simulations, with high and low sensor white noise, 10 iterations, very rare single-sensor faults (same configuration as the small simulation, which is discussed in section 5.6.4.1), and a variety of selected sensors from 2 to 32.

Experimental results reveal the following interesting findings:

a) the Cosine distance has the best average accuracy compared to the other histogram distances, which can be seen in figure 5.21. More research needs to be conducted to

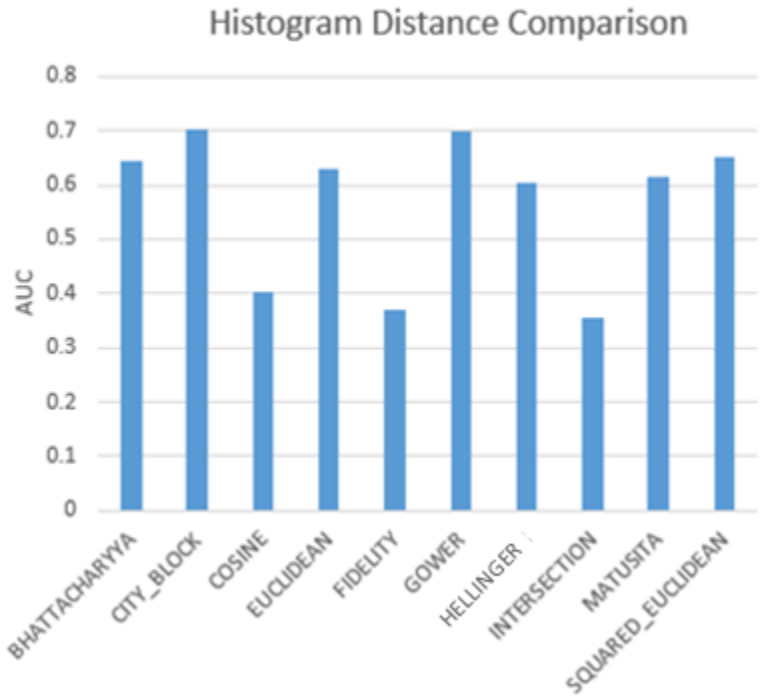

Figure 5.20: Comparison of histogram distances, when used in COSMO, using average AUC.

find out why Cosine slightly out-performs the other distances for larger simulations, although this is most likely the case since the Cosine distance is the only distance that contains a division. My hypothesis is that with a large fleet of vehicles, decreasing the model gathering period of 7 days could increase the accuracy of the results. [82] mention that for a small fleet, this parameter should be sufficiently large and for larger fleets, the parameter should be considerably smaller.

b) the average accuracy is generally lower for the histogram distances that perform reasonably well in smaller simulations. In other words, all histogram distances other than the Cosine, Fidelity, and Intersection distance perform more poorly in larger simulations.

c) the Intersection and Fidelity distances have a high standard deviation in their performance (see figure 5.22). At times these distances perform well, and at other times they perform quite poorly. This is interesting, since like Cosine, they perform poorly in the smaller simulations. On average they perform similarly to the other distance measures (other than Cosine) for larger simulations. Further research could be done to find out why the Fidelity and Intersection distance can perform quite well in these types of simulations.

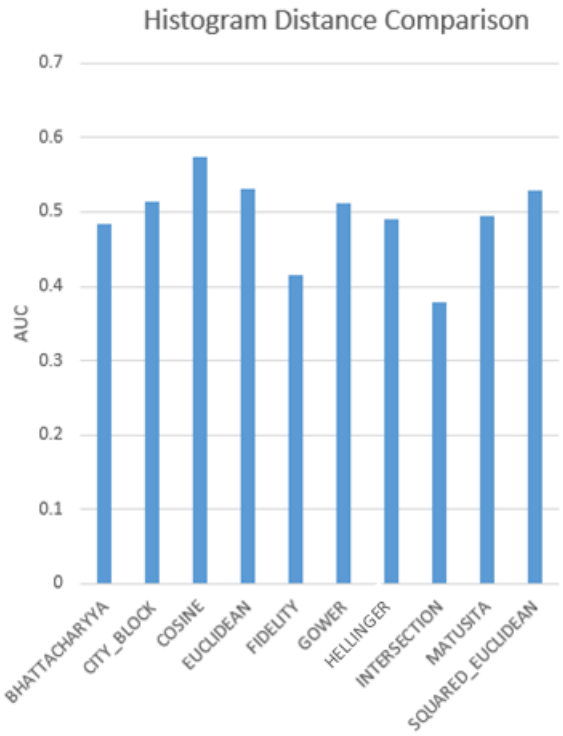

Figure 5.21: Comparison of histogram distances, when used in COSMO, using average AUC. High sensor noise and large simulation

5.6.5 Analyzing ICOSMO Parameter Effects

To get an idea of the effect the parameters have on the behavior of ICOSMO, I ran three Type B simulations to produce three different types of histories. Using each of these histories, I ran multiple Type A experiment sets. The parameters were modified slightly for each experiment in these sets. As such, analyzing the results of each experiment of the set would reveal the effect each parameter has on the results, since each experiment is run on the same history as the last. By increasing or decreasing a specific parameter, the effect of the parameter can be analyzed by plotting the average area under the curve (AUC) with respect to the experiment identification (for example, see figure 5.23). Note that with a constant history the only difference between simulation iterations is the non-determinism of the IRA when estimating fault-involved sensors.

5.6.5.1 Threshold Precision Experiment

A simulation's execution time is increased significantly when the simulation creates many unique z-scores, since these z-scores are used later in the simulation as thresholds to build ROC curve points. Therefore, reducing the number of thresholds should speed up the execution at the cost of the resulting accuracy, since there will be fewer ROC curve points created and the area under the ROC curves is used as the performance measure. In

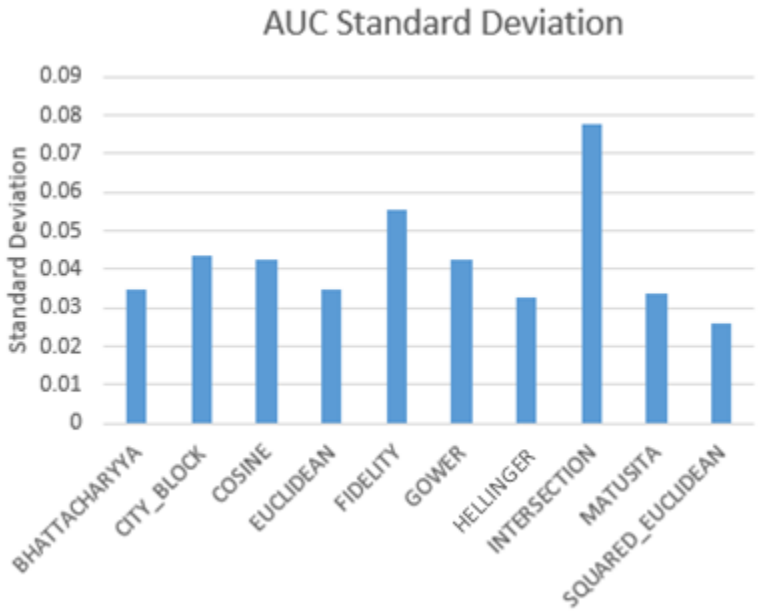

Figure 5.22: Comparison of histogram distances, when used in COSMO, using standard deviation of AUC. High sensor noise and large simulation.

other words, truncating all the thresholds to a defined precision will reduce the number of thresholds, which can be achieved by lowering the threshold decimal precision parameter. In this section, the accuracy of the simulation results is analyzed when varying the threshold decimal precision parameter (details of this parameter are discussed in the simulation logic part of section 5.3.7). The goal of this analysis is to discover characteristics about the accuracy versus speed trade-off.

I ran a Type A experiment set to see how much accuracy is lost when varying the threshold's decimal precision from 2, in E_1, to 7, in E_6. I ran a 60-day simulation, with 8 vehicles, 64 iterations, and 90% of the sensor readings removed (only 10% were sampled). I chose a Type A experiment, since the z-scores used to analyze the performance remain constant throughout the experiment set. I assume E_6, the experiment with the highest threshold decimal precision, produced the most accurate results, since it does not remove any unnecessary points from the ROC curves. Therefore, to compare each threshold decimal precision, I used the mean square error as follows,

$$\text{MSE} = \tfrac{1}{5}\textstyle\sum_{i=2}^{6}(AUC_7 - AUC_i)^2,$$

where AUC_i is the area under the ROC curve when using the threshold decimal precision $= i$.

The results are illustrated in figures A.12 and A.13, which reveal that there is only a fraction accuracy between low and high threshold decimal precision. This suggests running quick simulations with low threshold decimals precisions can be done without significantly

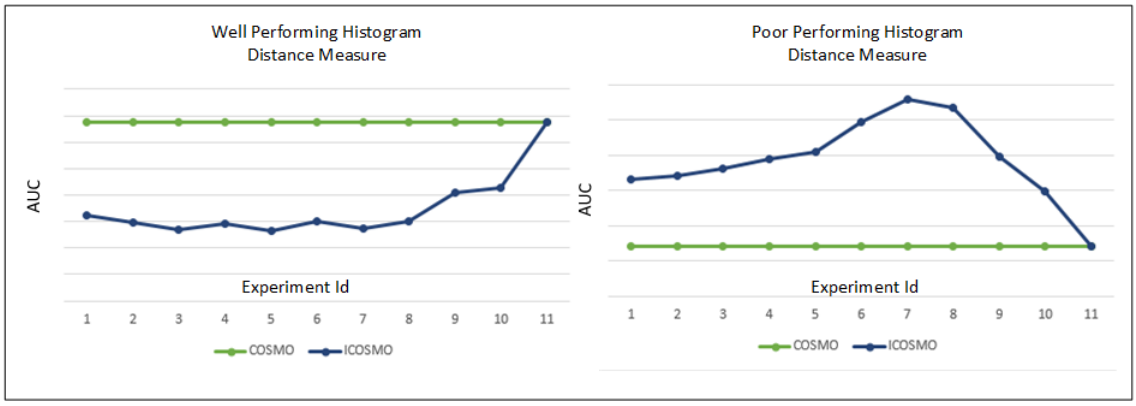

Figure 5.23: Experiment set 1. Average AUC performance, Type A experiment using H_1, desired precision 5%, increasing estimated fault-involved sensor and maximum added sensors parameters. Better performing histogram distances (Bhattacharyya, Hellinger, and Matusita) are on the left and the poor histogram distances (others) are on the right.

risking the accuracy of the results. It would have been interesting to analyze the execution time with respect to the threshold decimal precision, but due to time constraints such an analysis was not conducted.

It is worth noting that since ICOSMO has non-deterministic behavior in the simulation (the information retrieval algorithm, which is based on random list permutations), the difference in ICOSMO's performance results between the experiments cannot be 100% attributed to the change in threshold decimal precision. COSMO on the other hand will always produce identical results when running Type A experiment sets, so it can be deduced that the difference in COSMO's performance results between experiments can be 100% attributed to the change in threshold decimal precision.

5.6.5.2 Histories

This section discusses the configuration used to create the three histories used to run the Type A experiments. Let H_i denote the ith history. For H_1, the simulation ran for 60 days and it had 8 vehicles and 2 selected sensors. For the single-sensor failure fault attributes, the minimum number of days before repair was 5. The single sensor faults were very rare in this history, with occurrence rates from 0.0000001% to 0.000001%. The repair occurrence rates varied from 10% to 20%. Most of the sensors' behavior white noise parameters were configured such that their standard deviation, which were originally derived and assigned by analyzing the STO primary dataset (see section 5.6.3.2 for more details), were multiplied by a factor of 10 to simulate a high sensor noise environment. The amplification factor and noise amplification factor for most sensors was sampled from a normal distribution with mean = 1 and standard deviation = 0.05.

The other histories had similar configurations as H_1. The only difference between H_1 is that H_2 had 4 sensors selected. H_3 had 2 sensors selected as well, but it was a bit different. H_3 removed the added white noise by removing the multiplication factor from the sensor

History 1		History 2		History 3	
Dist.	**AUC**	**Dist.**	**AUC**	**Dist.**	**AUC**
Matusita	0.92	Fidelity	0.53	Matusita	0.92
Hellinger	0.86	Gower	0.50	Hellinger	0.86
Bhatta.	0.85	Squared Euc.	0.50	Bhatta.	0.83
City Block	0.69	City Block	0.50	City Block	0.75
Gower	0.68	Matusita	0.48	Gower	0.74
Squared Euc.	0.64	Hellinger	0.48	Euclidean	0.69
Euclidean	0.60	Euclidean	0.48	Squared Euc.	0.65
Cosine	0.24	Bhatta.	0.45	Intersection	0.32
Intersection	0.16	Intersection	0.43	Cosine	0.20
Fidelity	0.07	Cosine	0.39	Fidelity	0.15

Table 5.6: Performance of the COSMO approach when using various histogram distances for H_{1-3}, where Dist. = Distance and AUC = Area Under the Curve.

behavior white noise parameters; the minimum days before repair for the single-sensor failures was 10 instead of 5, and the single-sensor failure fault occurrence rates varied from 0.1 to 1% instead of being very rare. Note that for notation purposes, H_{1-2} will be referred to as *high noise* histories and H_3 will be referred to as a *low noise* history. Table 5.6 summarizes the AUC of the COSMO distances for each history.

5.6.5.3 Poor Precision - Stale Sensor Removal Only - Experiment Set 1

This is a Type A experiment set that uses H_1 as its history. In each of its experiments E_1 to E_{11}, the number of estimated fault-involved sensors is increased from 13, in E_1, to 60, in E_{11}, and the number of maximum sensors to add is increased from 3, in E_1, to 60, in E_{11}. The other ICOSMO parameters remain constant for the experiment set and are defined as follows: desired precision = 5%, deviation search window = 0, staleness threshold = 4, candidacy threshold = 6, SC decrease modifier = 5, SC increase modifier = 1, SPC decrease modifier = 5, SPC increase modifier = 1, default SC = 5, default SPC = 5, and maximum number of removed sensors = 1. This parameter configuration ensures that ICOSMO will most likely never add a candidate sensor to the COSMO sensors. Its only option is to remove a single sensor from the COSMO sensors, since H_1 only has 2 sensors selected.

Experimental results, which are illustrated in figure 5.23, reveal the following interesting findings:

a) ICOSMO decreased the accuracy of COSMO for the Bhattacharyya, Hellinger, and Matusita distances (see the left chart in figure 5.23), which are the better performing distances in H_1 (see Column 1 of Table 5.6), and ICOSMO increased the accuracy of the other distances (see the right chart in figure 5.23), which perform more poorly than histogram distances in H_1.

b) ICOSMO's only option for changing the sensor selection is removing a single sensor from the COSMO sensors (one of the two sensors selected), since, a) the desired precision is 5% precision, and as a result the sensor ranking adjustment is essentially random; and b), ICOSMO's configuration will likely only flag sensors as candidates, since the SPC decrease modifier is so strict. Therefore, since ICOSMO's only option is to remove a sensor from the COSMO sensors, the following can be deduced:

   i) in H_1, the Bhattacharyya, Hellinger, and Matusita distance chose their best sensors, since each of their accuracies is lowered on average by ICOSMO when ICOSMO removes an arbitrary sensor from their COSMO sensors.

   ii) on the other hand in H_1, the distances other than the above have each failed to choose their best sensors, since each of their accuracies is increased on average by ICOSMO when ICOSMO removes an arbitrary sensor from their COSMO sensors.

c) the reason ICOSMO's accuracy converges toward COSMO's accuracy (for all histogram distances) as the experiment number increases is not clear. More research is required to investigate the cause of this convergence.

5.6.5.4 Poor Recall - Stale Sensor Removal Only - Experiment Set 2

This is a Type A experiment set that uses H_1 as its history. The desired recall is increased from 10%, in E_1, to 85%, in E_{11}. The other ICOSMO parameters remain constant for the experiment set and are defined as follows: deviation search window = 0, staleness threshold = 4, candidacy threshold = 6, SC decrease modifier = 5, SC increase modifier = 1, SPC decrease modifier = 5, SPC increase modifier = 1, default SC = 5, default SPC = 5, number of estimated fault-involved sensors = 30, maximum number of added sensors = 48, and maximum number of removed sensors = 1. This parameter configuration ensures that ICOSMO will most likely never add a candidate sensor to the COSMO sensors. Its only option is to remove a single sensor from the COSMO sensors, since H_1 has only 2 sensors selected.

Experimental results, which are illustrated in figure 5.24, reveal the following interesting findings:

a) the results of this experiment set are similar to experiment set 1 (see section 5.6.5.3). That is, ICOSMO decreased the accuracy of COSMO for the Bhattacharyya, Hellinger, and Matusita distance, since they chose the best sensors, and ICOSMO increases the accuracy of the other distances, since they did not choose the best sensor selection.

b) the difference between this experiment set and experiment set 1 is that the ICOSMO accuracy generally remains constant for all experiments in the set, while in experiment set 1 the accuracy is not constant (see figure 5.23).

   This can be explained by the parameter configuration of this experiment set. Referring to equation 4.8, to meet the desired precision accurately when generating the

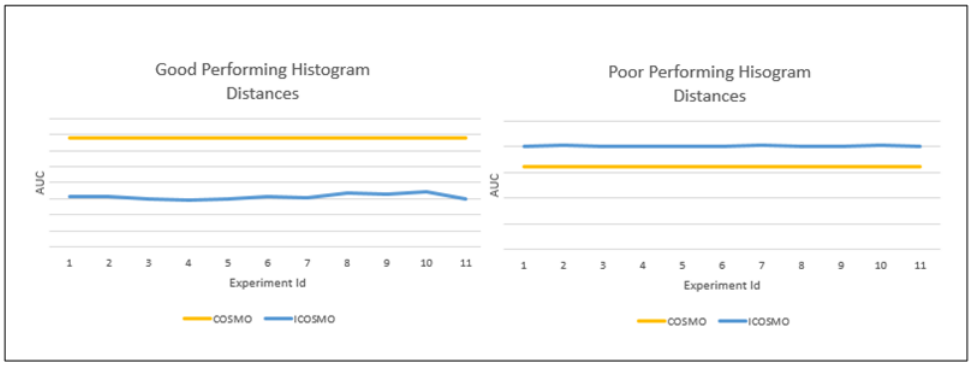

Figure 5.24: Experiment set 2. Average AUC performance, Type A experiment using H_1, desired recall is increased from 10% to 85%. Better performing histogram distances (Bhattacharyya, Hellinger, and Matusita) are on the left and the poor histogram distances (others) are on right.

set S, it depends on the number of fault-involved sensors being estimated, n. By increasing n, S will more closely have the desired precision (which is done in experiment set 1). However, when recall is desired, referring to equation 4.9, meeting the accuracy of the recall desired for the set S depends on the number sensors that are affected by a randomly occurring fault, $|P|$. Note that $|P|$ remains a constant number on average in this experiment set, since multiple iterations of the simulations are run, and the faults, which define P, have a constant occurrence rate throughout the experiment set. Therefore, ICOSMO produces relatively constant results because, even though the desired recall is being increased, $|P|$, which is constant on average, has more influence on affecting the value of the variable i from equation 4.9 than the desired recall does. This means i remains constant on average throughout the experiment set, and as a result ICOSMO behaves similarly for each experiment.

c) it would be interesting to investigate the effect of adding faults that have many fault-involved sensors, when running experiments that vary the desired recall of the IRA. For example, from E_1 to E_{11}, the probability of a fault with large P increases.

5.6.5.5 Decent Precision - Generous Candidate Sensor Addition - Experiment Set 3

This is a Type A experiment set that uses H_2 as its history. The SPC increase modifier is increased from 5, in E_1, to 15, in E_{11}, and then it is increased from 15, in E_{12}, to 25, in E_{22}. The other ICOSMO parameters remain constant for the experiment set and are defined as follows: deviation search window = 0, staleness threshold = 4, candidacy threshold = 6, SC decrease modifier = 2, SC increase modifier = 1, SPC decrease modifier = 1, desired precision = 70%, default SC = 5, default SPC = 5, number of estimated fault-involved sensors = 11, maximum number of added sensors = 8, and maximum number of removed sensors = 7. This parameter configuration favors adding candidate sensors.

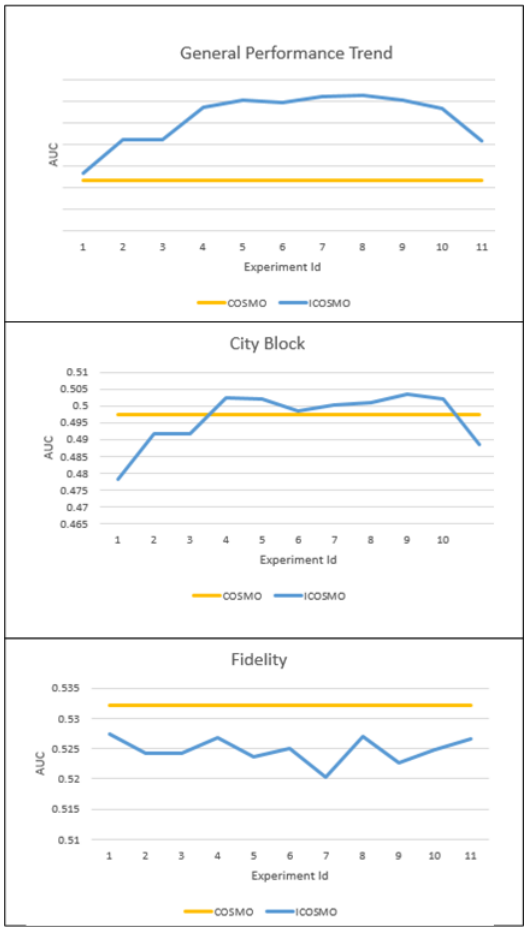

Figure 5.25: Experiment set 3. Average AUC performance, Type A experiment using H_2, where *General Performance Trend* is the general behavior of all the histogram distances other than the City Block and Fidelity distances.

Experimental results, which are illustrated in figure 5.25, which illustrates the behavior for E_{1-11}, and figure 5.26, which illustrates the average behavior of all histogram distances for E_{12-22}, reveal the following interesting findings:

a) referring to figure 5.25, from E_4 to E_{10} on average, ICOSMO increases the AUC for all histogram distance by 2-3% (for H_2's AUC performance, see Table 5.6). City Block slightly under-performs and Fidelity does worse than COSMO for the average trend for experiment E_1 to E_{11}.

b) E_{11} generally performs worse than E_{10} for all histogram distances except Fidelity.

c) from E_{12} to E_{22}, the accuracy remains relatively constant and converges. The convergence is 0.5-1% below the accuracy of ICOSMO for the experiments E_4 to E_{10} (see figure 5.26 for the general AUC trend for E_1 to E_{24}).

d) the first spike in figure 5.26, which occurs at SPC increase modifier 7, is assumed to be the result of the first candidate sensor being added to the COSMO sensors.

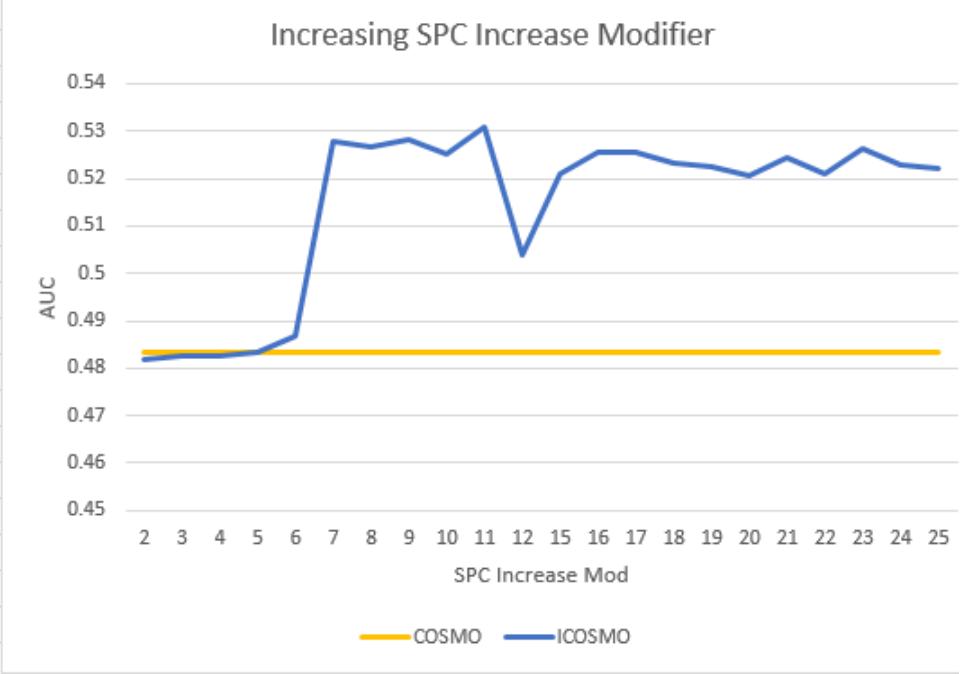

Figure 5.26: Experiment set 3. Type A experiment set on H_2. The average area under the curve (y-axis) for all histogram distances with respect to the potential contribution increase modifier (x-axis), with 70% desired precision.

Before this point ICOSMO probably did not flag any sensor as a candidate sensor, since the SPC increase modifier was not high enough.

e) it would be interesting to investigate why for all histogram distances there is an average AUC decrease when the SPC increase modifier = 12.

f) it would be interesting to conduct further research to identify how many candidate sensors added optimizes the accuracy, since there is a limit to how many sensors can be added in an IoT environment.

5.6.5.6 Decent Precision - Generous Candidate Sensor Addition and Stale Sensor Removal - Experiment Set 4

This is a Type A experiment set that uses H_2 as its history. The SPC increase modifier is increased from 2, in E_1, to 12, in E_{11}. The other ICOSMO parameters remain constant for the experiment set and are defined as follows: deviation search window = 0, staleness threshold = 4, candidacy threshold = 6, SC decrease modifier = 8, SC increase modifier = 2, SPC decrease modifier = 1, desired precision = 70%, default SC = 5, default SPC = 5, number of estimated fault-involved sensors = 11, maximum number of added sensors = 8, and maximum number of removed sensors = 7. This type of configuration is similar to the configuration in experiment set 3 (see section 5.6.5.5), but it generously removes stale sensors in addition to adding candidate sensors.

Experimental results, reveal the following interesting findings:

a) **Finding a)** for all histogram distances (except Fidelity, which is the exception and does relatively poorly), the SPC increase modifier = 7 (E_6) yields the best results. All the histogram distances gain 3-5% accuracy from ICOSMO (see Column 2 of Table 5.6), which is slightly better than the results in experiment set 3.

b) **Finding b)** after experiment E_6, the accuracy gradually decays for all the histogram distances.

5.6.5.7 Decent Precision - Low Noise - Experiment Set 5

This is a Type A experiment set that uses H_3 as its history. The SC increase modifier is increased from 3, in E_1, to 7, in E_5. The other ICOSMO parameters remain constant for the experiment set and are defined as follows: deviation search window = 3, staleness threshold = 4, candidacy threshold = 6, SC decrease modifier = 7, SPC decrease modifier = 5, SPC increase modifier = 7, desired precision = 70%, default SC = 5, default SPC = 5, number of estimated fault-involved sensors = 32, maximum number of added sensors = 8, and maximum number of removed sensors = 1.

Results reveal that ICOSMO has no effect on the performance of COSMO. This is interesting, since in all other experiment sets ICOSMO was able to affect the accuracy of COSMO, but in the case of H_3, which has low sensor noise, ICOSMO has no effect on COSMO. My hypothesis for this behavior is that since ICOSMO by design assumes COSMO selects poor sensors and attempts to improve COSMO, because there is low sensor noise, COSMO may have chosen the best two sensors in H_3, and therefore, ICOSMO cannot improve the results.

5.6.6 ICOSMO vs. COSMO

5.6.6.1 Large Simulation - High Sensor Noise - Low Precision - Experiment Set 6

This is a Type B experiment set, and the simulation and COSMO are configured as follows: 250-day simulation length, 32 vehicles, 4 select sensors for 4 simulation iterations and 5 selected sensors for 2 simulation iterations, high sensor white noise, and very rare single-sensor faults (0.0000001% to 0.00001% occurrence rates, 10% to 30% repair occurrence rates, and a minimum of 2 days before repaired). ICOSMO was configured as follows, deviation search window = 0, staleness threshold = 3.75, candidacy threshold = 7.5, SC decrease modifier = 1, SC increase modifier = 1, SPC decrease modifier = 1, SPC increase modifier = 1, desired precision = 10%, default SC = 5, default SPC = 5, number of estimated fault-involved sensors = 16, maximum number of added sensors = 8, and maximum number of removed sensors = 3, for 4 iterations and, the maximum number of removed sensors = 4 for 2 iterations. This type of configuration favors flagging sensors as stale,

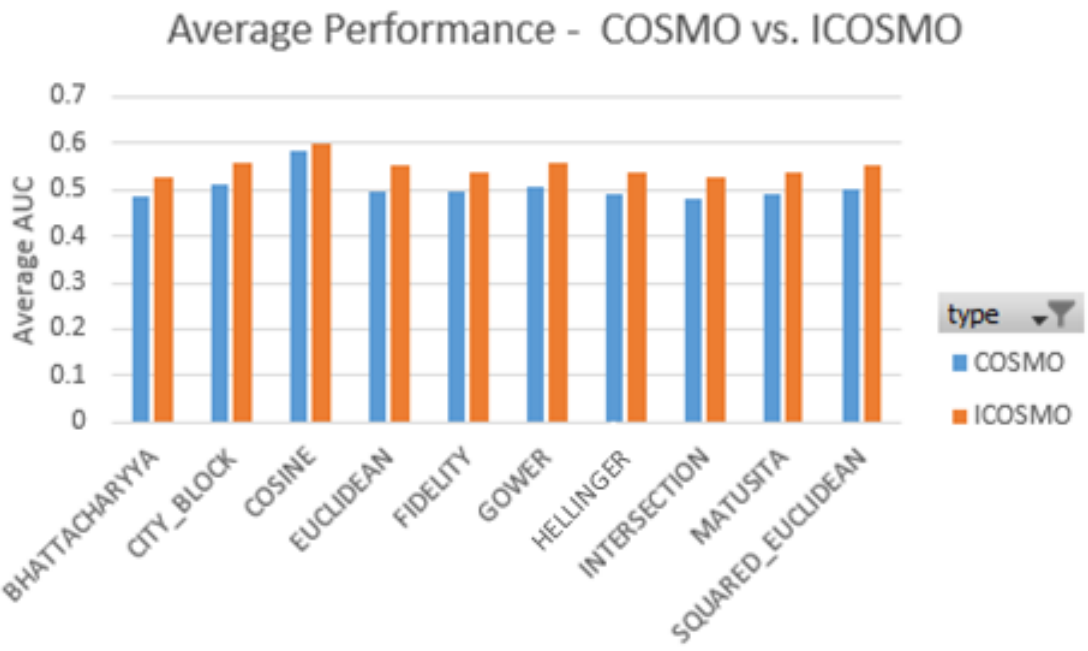

Figure 5.27: Experiment set 6. , ICOSMO vs. COSMO average AUC performance, Type B experiment.

since the staleness threshold is more relaxed than the candidacy threshold. Note that only a few iterations were run, since large iterations take more time to execute.

Results reveal, which are illustrated in figure 5.27, the following interesting findings:

a) the Cosine distance out-performs the other histogram distances. It is unclear why Cosine does significantly better in large simulations than in smaller simulations (8 vehicles and 60-day simulations).

b) ICOSMO out-performs COSMO for all histogram distances by 4-5%, except in the case of the Cosine distance. I believe this is the case because it is more likely that sensors be flagged as stale than as candidates. That is, eventually more COSMO sensors will be poorly flagged as stale sensors than non-COSMO sensors flagged as candidates, which means that on multiple occasions sensors will be considered stale, even when they are not. This means the only effect ICOSMO has is when it removes one of the poorly flagged stale COSMO sensors. Since there is a lot of noise in sensor data generation, removing a sensor most likely means better decisions will be made, since fewer noisy sensors are selected, and ICOSMO therefore performs more accurately. With low noise sensors, I do not think ICOSMO would increase the average AUC by 5% for most of the histogram distances.

c) ICOSMO only increases the average AUC of COSMO by 1% for the Cosine distance. This may be explained by the finding that the better COSMO performs, the harder it is for ICOSMO to increase the performance of COSMO.

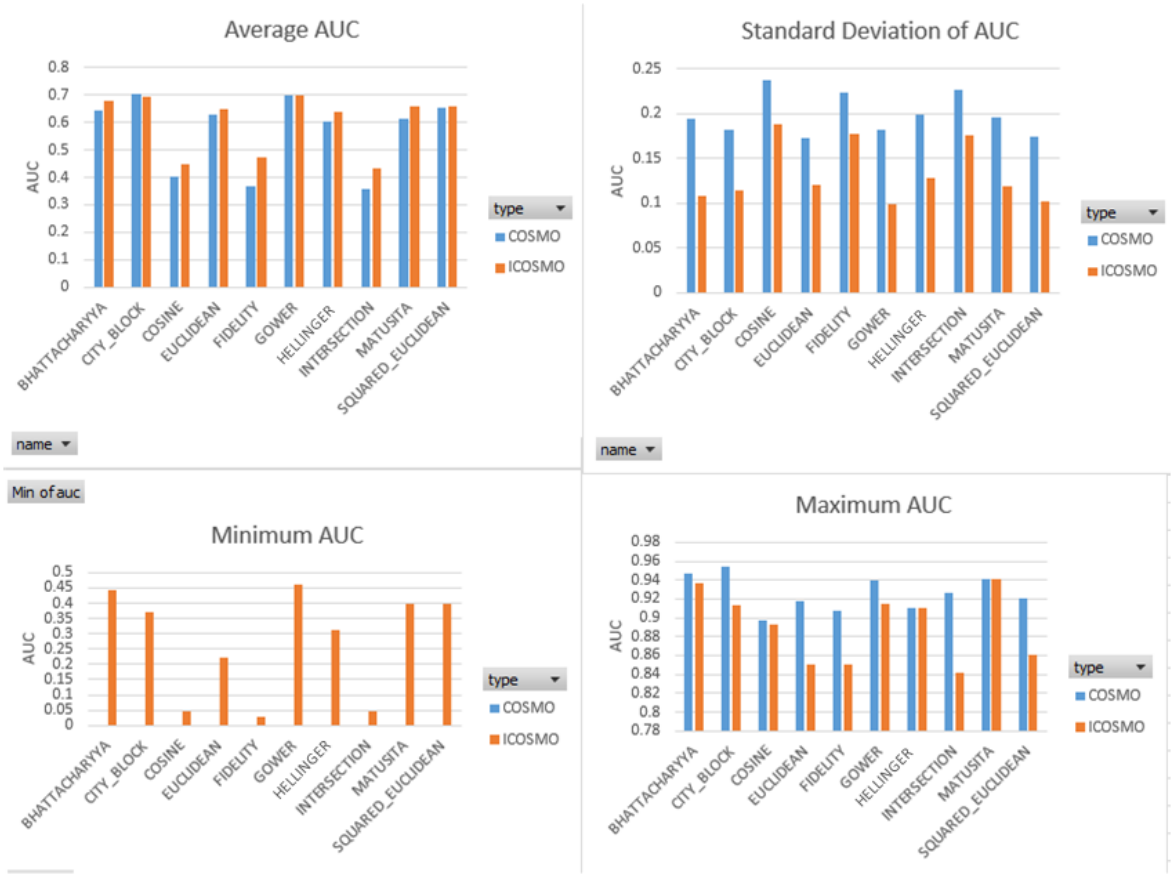

Figure 5.28: Experiment set 7. Example of ICOSMO improving the accuracy (average AUC) of poor COSMO sensor selection choices.

5.6.6.2 Small Simulation - High Sensor Noise - Decent Precision - Experiment Set 7

This is a Type B experiment set, and the simulation and COSMO are configured as follows: 60-day simulation length, 8 vehicles, 2 select sensors, with 43 iterations, with high sensor white noise, and very rare single-sensor faults (0.0000001% to 0.000001% occurrence rates, 10% to 20% repair occurrence rates, and a minimum of 5 days before repaired). ICOSMO was configured as follows: deviation search window = 3, staleness threshold = 4, candidacy threshold = 6, SC decrease modifier = 20, SC increase modifier = 2, SPC decrease modifier = 1, SPC increase modifier = 7, desired precision = 70%, default SC = 5, default SPC = 5, number of estimated fault-involved sensors = 11, maximum number of added sensors = 8, and maximum number of removed sensors = 1. This type of configuration should flag and remove stale sensors quickly, and it should flag and add candidate sensors quickly.

Results reveal, which are illustrated in figure 5.28 and figure 5.29, the following interesting findings:

a) ICOSMO significantly improves the accuracy of the COSMO approach when the sensors selected by COSMO were a poor choice. By referring to the bottom-left

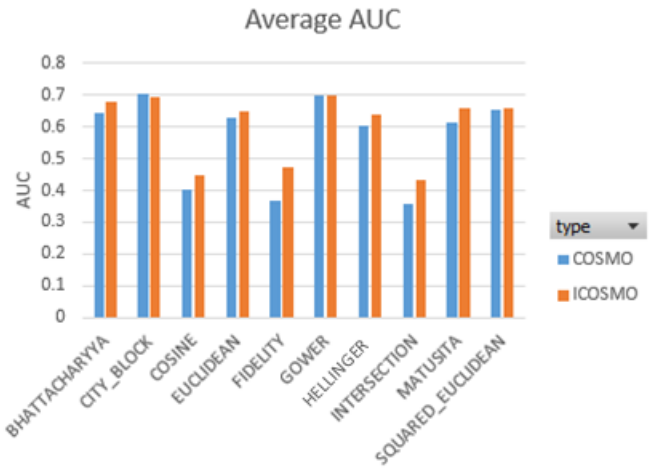

Figure 5.29: Experiment set 7. Example of ICOSMO improving the accuracy (average AUC) of poor histogram choices.

"Minimum AUC" chart in figure 5.28, the minimum AUC for all histogram distances was 0.00 for COSMO, while there are no such occurrences for ICOSMO. In fact, ICOSMO's minimum AUC values range from 20 to 44% for the histogram distances, other than Fidelity, Cosine, and Intersection. In other words, for better performing histogram distances, ICOSMO significantly improves the performance of COSMO when COSMO is doing poorly. However, it should be noted that most likely COSMO did not actually fail to predict any positive readings for some simulation iterations. The reason for this behavior is most likely because faults involving only the two selected sensors never occurred due to the probabilistic nature of the simulation, so it was impossible for COSMO to predict a positive reading. Nevertheless, it is still interesting that ICOSMO was able to increase the accuracy from 0% to 44% by adding candidate sensors.

b) on average ICOSMO increased most significantly the performance of the Fidelity and Intersection distance by 11% and 8%, respectively, compared to the other histogram distances (see figure 5.28, top-left "Average AUC" chart). Both these distances yielded the worst average AUC for the COSMO approach. Other experimental results support the finding that, in general, when a histogram distance is doing relatively poorly, ICOSMO significantly improves the accuracy of the COSMO approach. Figure 5.29 illustrates a result set where ICOSMO adds more accuracy to the under-performing histogram distances. However, in some cases ICOSMO does not significantly improve COSMO when COSMO is doing poorly; for example, illustrated in the bottom-left "Minimum AUC" chart of figure 5.28, the Cosine, Fidelity, and Intersection distance do not benefit much from ICOSMO. Finding *a)* above suggests that when COSMO fails to detect all faults using a poor histogram distance, ICOSMO will increase the accuracy of the **better** performing histogram distances, while the **under-performing** distance will barely be improved by ICOSMO.

c) the maximum AUC values for COSMO are greater than or equal to the maximum AUC values for ICOSMO (see figure 5.28, bottom-right "Maximum AUC" chart). This suggests ICOSMO generally only improves COSMO when COSMO is performing poorly. My explanation for this behavior is that ICOSMO attempts to improve the accuracy of COSMO, with the assumption that COSMO did not choose the best COSMO sensors. ICOSMO must observe NFFS (or false negatives) before adjusting the sensor selection. In other words, poor predictions are required by ICOSMO, which decrease performance results temporarily, before ICOSMO can begin to benefit COSMO's accuracy. This means when COSMO has made the best sensor selection choice, ICOSMO cannot produce better AUC results.

d) COSMO, on average, out-performed ICOSMO for the City Block distance by 0.7%. This is interesting, since in general either ICOSMO has no effect on COSMO or it increases COSMO's performance. Further research is required to explain this behavior.

5.6.6.3 Small Simulation - High Sensor Noise vs. Small Sensor Noise - Experiment Set 8

In this experiment set, there are two types of experiments that are run: a) experiments with high sensor noise and b) experiments with low sensor noise. The goal of this experiment set is to investigate the effects of sensor noise by comparing the results of both categories of experiments. In this Type B experiment set, the experiment set is configured as follows:

- **High noise** the simulation and COSMO are configured as follows: 60-day simulation length, 8 vehicles, 2 select sensors, 64 iterations, high sensor white noise, and common single-sensor faults (0.01% to 0.1% occurrence rates, 10% to 20% repair occurrence rates, and a minimum of 10 days before repaired). ICOSMO was configured as follows: deviation search window = 3, staleness threshold = 4, candidacy threshold = 6, SC decrease modifier = 20, SC increase modifier = 2, SPC decrease modifier = 1, SPC increase modifier = 7, desired precision = 70%, default SC = 5, default SPC = 5, number of estimated fault-involved sensors = 11, maximum number of added sensors = 8, and maximum number of removed sensors = 1.

- **Low noise** the simulation and COSMO are configured as follows: 60-day simulation length, 8 vehicles, 2 select sensors, 104 iterations, high sensor white noise, and common single-sensor faults (0.01% to 0.1% occurrence rates, 10% to 20% repair occurrence rates, and a minimum of 10 days before repaired). ICOSMO was configured as follows: deviation search window = 3, staleness threshold = 4, candidacy threshold = 6, SC decrease modifier = 1, SC increase modifier = 1, SPC decrease modifier = 1, SPC increase modifier = 12, desired precision = 70%, default SC = 5, default SPC = 5, number of estimated fault-involved sensors = 32, maximum number of added sensors = 8, and maximum number of removed sensors = 1. This type of configuration was put in place to favor adding candidate sensors, since other experiments revealed that a high SPC increase modifier benefits the performance in noisy sensor simulations.

The following abbreviations and definitions are used in this section to facilitate the analysis:

- *IH* ICOSMO with high sensor noise
- *IL* ICOSMO with low sensor noise
- *CH* COSMO with high sensor noise
- *CL* COSMO with low sensor noise
- *poor histogram distances* the Fidelity, Cosine, and Intersection distances
- *decent histogram distances* all the histogram distances that are not the Fidelity, Cosine, or the Intersection distance

The result comparison of both types of experiments, which is illustrated in figure A.6, reveals the following interesting findings:

a) the poor histogram distances are affected by sensor noise more significantly than the other distances. That is, $|CH - CL|$ and $|IH - IL|$ are greater for the poor histogram distances than for the other distances.

b) IH outperforms IL, CH, and CL for all the histogram distances except the poor histogram distances. In fact, the opposite is true for the poor histogram distances; IL and CL outperform IH and CH.

c) for histogram distances City Block and Gower, CH performs slightly better than CL. For the other decent histogram distances the opposite is true; CH performs slightly more poorly than CL.

5.6.7 Summary

All the experiments performed in this work reveal that, in general, ICOSMO improves the accuracy of COSMO when COSMO is not performing optimally; that is, in the following situations: a) when the histogram distance chosen by COSMO is a poor choice, b) in an environment with relatively high sensor white noise, and c) when COSMO selects poor sensors. ICOSMO only rarely reduced the accuracy of COSMO on average, which is promising, since it suggests deploying ICOSMO as a predictive maintenance system should perform just as well as or better than COSMO.

ICOSMO tends to improve the accuracy of COSMO more significantly when the histogram chosen is a poor choice, and tends to improve the accuracy only slightly when the histogram distance chosen is a good choice. This is most likely the case, since the results revealed that typically when COSMO is doing well, ICOSMO does not improve its accuracy significantly (or at all). For small simulations with high sensor noise, all the

histogram distances, except the Fidelity, Intersection, and Cosine distance (which do more poorly), tend to perform similarly. Therefore, when choosing the histogram distance, the distance can be chosen based on its computational complexity, since they all generally perform the same. In larger simulations with high sensor noise, the *Cosine* distance is the best performing histogram distance.

When ICOSMO improves the accuracy most significantly, ICOSMO's parameters are configured such that many candidate sensors can be added to the selected sensors. Care must be taken, though, since in IoT not all sensors can be included in the model. Accuracy improvements were also obtained when configuring ICOSMO to favor removing stale sensors. This is promising, since it suggests that deploying sensor installation suggestions (see section 4.8.2 for more details) would benefit ICOSMO.

There is still more research to be made, since in some experiments there was no obvious answer to make sense of some results. It would also be interesting to include a different methodology for counting the TP, FP, TN, and FN when evaluating ICOSMO and COSMO. At the moment, faults that involve non-COSMO sensors do not affect the TP, FP, TN, and FN rates. Instead of only considering the COSMO sensors in the calculation, it would be insightful to also consider the non-COSMO sensors.

Chapter 6

Conclusion

The present work proposes a novel IoT-based architecture for predictive maintenance. The architecture consists of three primary nodes: the vehicle node (VN), the server leader node (SLN), and the root node (RN). The VN represents the vehicle, and performs lightweight data acquisition, data analytics, and data storage. The VN is connected to the fleet via its wireless internet connection. The SLN is responsible for managing a region of vehicles, and it performs more heavy-duty data storage, fleet-wide analytics, and networking. The RN is the central point of administration for the entire system. It controls the entire fleet and provides the application interface to the fleet system. A minimally viable prototype (MVP) of the proposed architecture was implemented and deployed to a garage of the Société de Transport de l'Outaouais (STO), Gatineau, Canada. The VN in the MVP was implemented using a Raspberry Pi, which acquired sensor data from a STO hybrid bus by reading from a J1939 network. The SLN was implemented using a laptop. The RN was deployed using `meshcentral.com`. The goal of the MVP was to perform predictive maintenance for the STO to help reduce their fleet management costs. Limitations of the MVP include; a) only a single VN was deployed at the STO, b) the J1939 data acquired was not labeled, which means fault was unavailable, and c) no intelligence was deployed to the MVP; it was only used for data acquisition.

The present work also proposes a fleet-wide unsupervised dynamic sensor selection algorithm, which attempts to improve the sensor selection performed in the Consensus Self-organized Models (COSMO) approach from [82], which I named the Improved Consensus Self-organized Models (ICOSMO) approach. To analyze the performance of ICOSMO, a fleet simulation was implemented. The J1939 data gathered from a STO hybrid bus, which was acquired using the MVP, was used to generate synthetic data to simulate vehicles, faults, repairs. The deviation detection of the COSMO and ICOSMO approach were applied to the synthetic sensor data. The simulation results were used to compare the performance of the COSMO and ICOSMO approach. One of the limitations of ICOSMO is it cannot be deployed to the MVP, since its assumptions are not met. ICOSMO's unmet assumptions are as follows: a) access to a vehicle service record database (VSRDB) is available (but the STO would not provide it to us) and b) access to an information retrieval algorithm (IRA) that estimates fault-involved sensors is not possible, as such an IRA does not exist.

The future work of this thesis includes the following: a) injecting faults into a bus to gather fault data, b) deploying predictive analytic algorithms to the MVP, c) deploying additional VNs to the MVP, d) hosting the RN on a private server instead of `meshcentral.com`, e) gathering more J1939 data from the MVP, f) evaluating the performance of COSMO and ICOSMO using exclusively labeled live data, which would require labeled fault data, instead of using synthetic data, g) replacing the IRA and VSRDB in ICOSMO by on-board predictive analytics algorithms, which would detect faults and identify fault-involved sensors, h) running more simulations on different J1939 STO datasets, i) performing concept drift research and applying it to ICOSMO, and j) making ICOSMO more stable; that is, although ICOSMO will only adjust the ranking (SC and SPC) of the sensor instances when a repair occurs (which renders the stability of ICOSMO directly proportional to the frequency of repairs), it is still possible that ICOSMO becomes unstable. In Algorithm 5, there are no bounds on the SPC and SC, which means that they could tend toward extreme values and ICOSMO could no longer recover from such a state. Therefore, a moving average window and/or bounding these values could solve this issue.

APPENDICES

Appendix A

Results and Diagrams

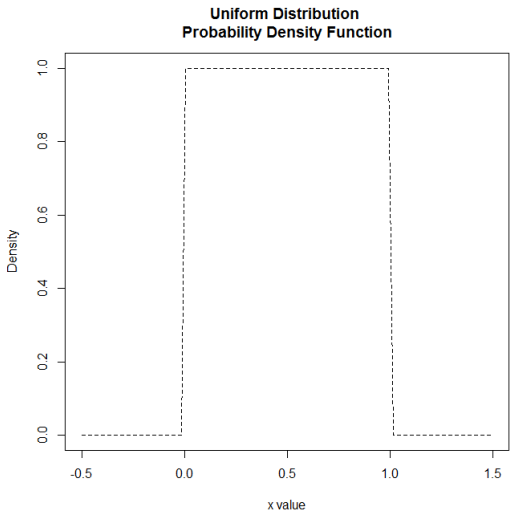

Figure A.1: Example of a probability density function for a uniform distribution with minimum = 0 and maximum = 1.

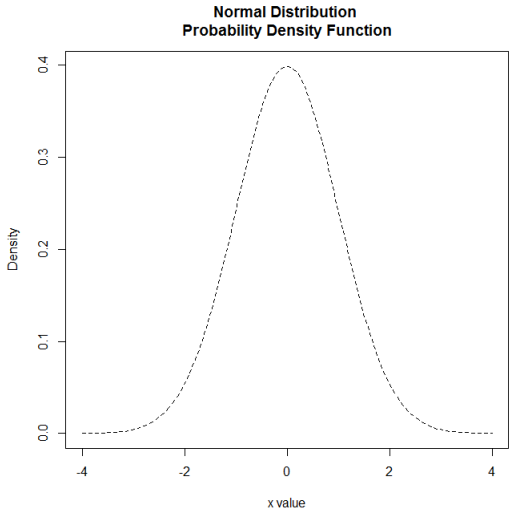

Figure A.2: Example of a probability density function for a normal distribution with mean = 0 and standard deviation = 1.

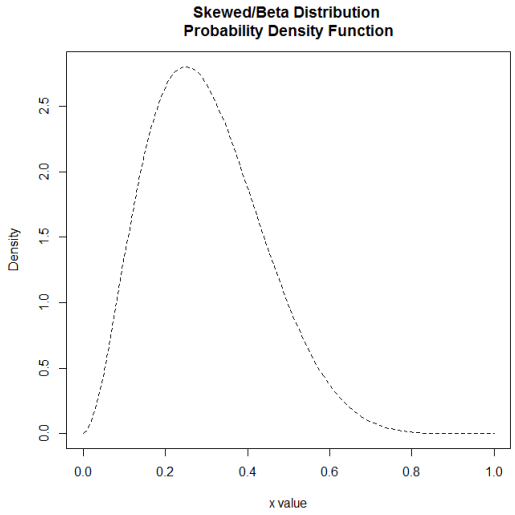

Figure A.3: Example of a probability density function for a skewed/beta distribution, where shape1 = 3, and shape2 = 7.

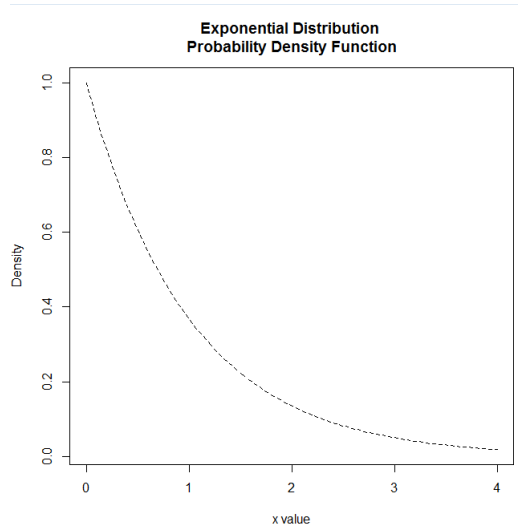

Figure A.4: Example of a probability density function for an exponential distribution.

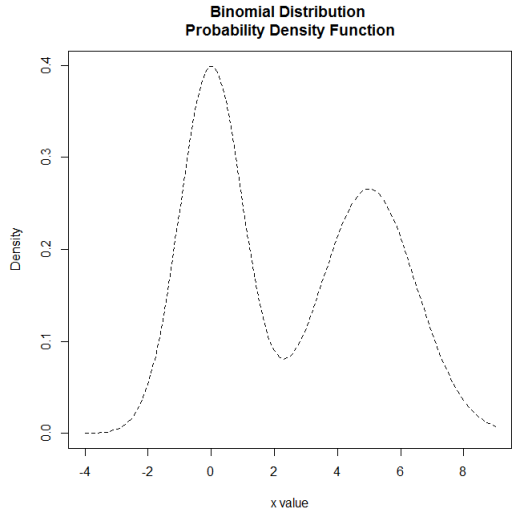

Figure A.5: Example of a probability density function for a Bimodal distribution. It is composed of 2 normal distributions. The first normal distribution nas mean = 0 and standard deviation = 1, and the second normal distribution has mean = 5 and standard deviation = 1.5.

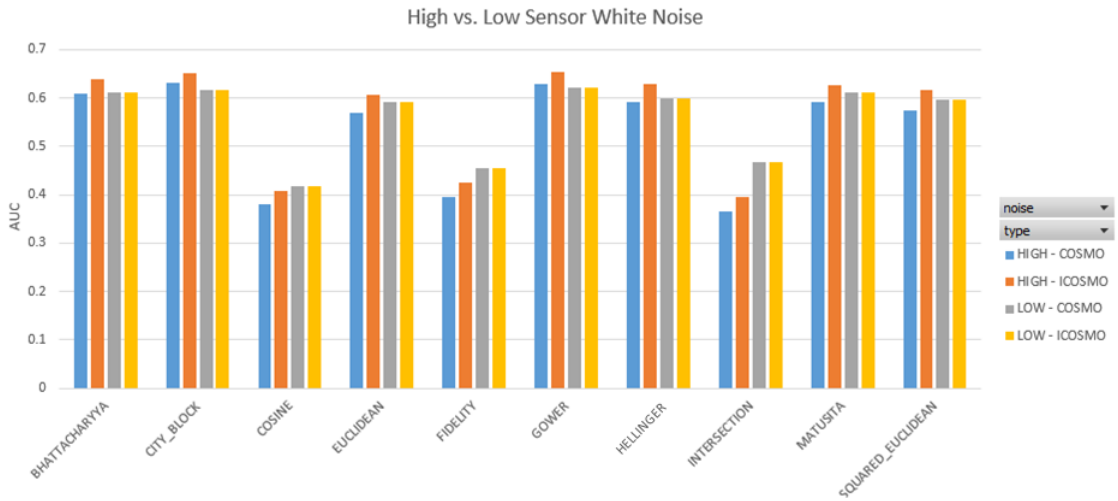

Figure A.6: Experiment set 8. Comparing the average AUC of COSMO and ICOSMO with high and low sensor white noise.

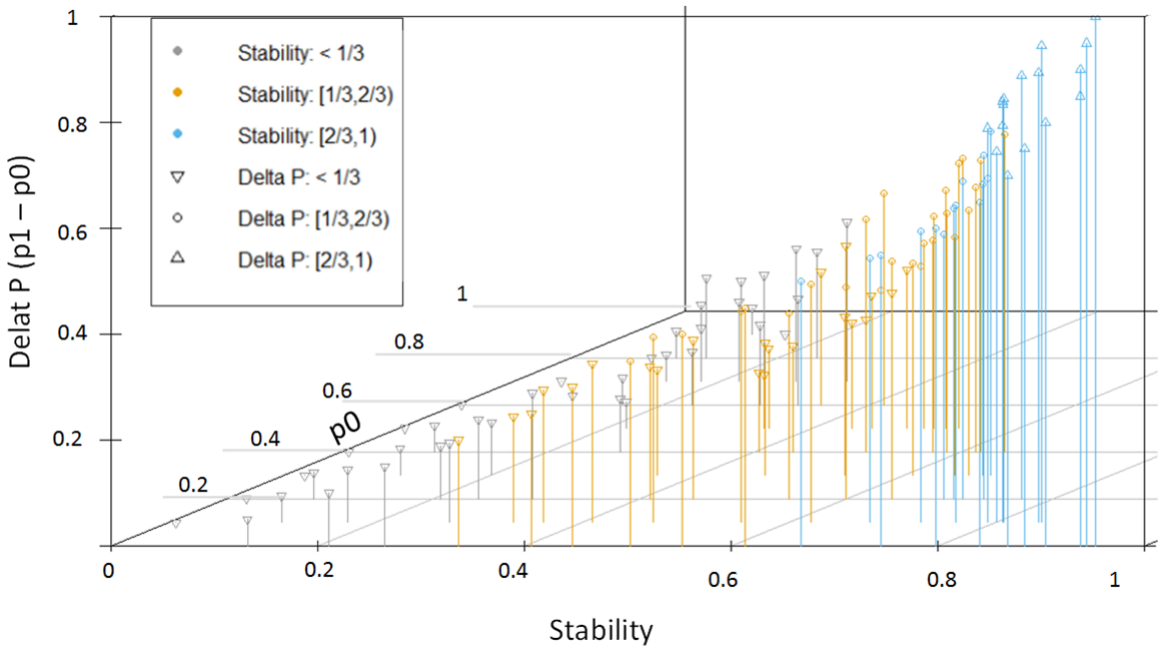

Figure A.7: Stability analysis, experiment 1: stability vs. Δp_x values when generating synthetic data using Equation 5.1, using uniformly distributed white noise. The original dataset is illustrated in Figure A.9.

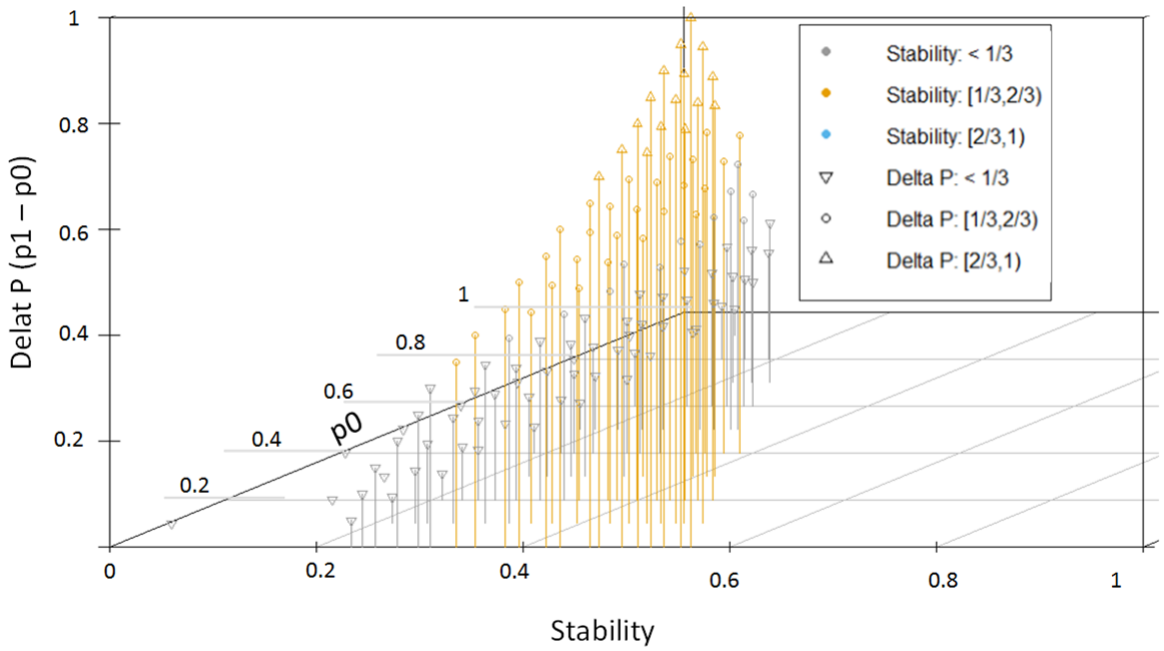

Figure A.8: Stability analysis, experiment 2: stability vs. Δp_x values when generating synthetic data using Equation 5.1, using normally distributed white noise. The original dataset is illustrated in Figure A.9

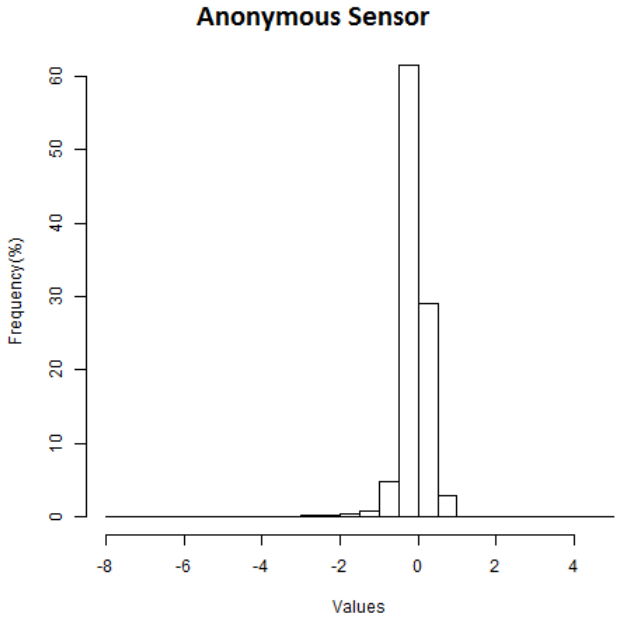

Figure A.9: Histogram of anonymous STO sensor data with 215,348 samples.

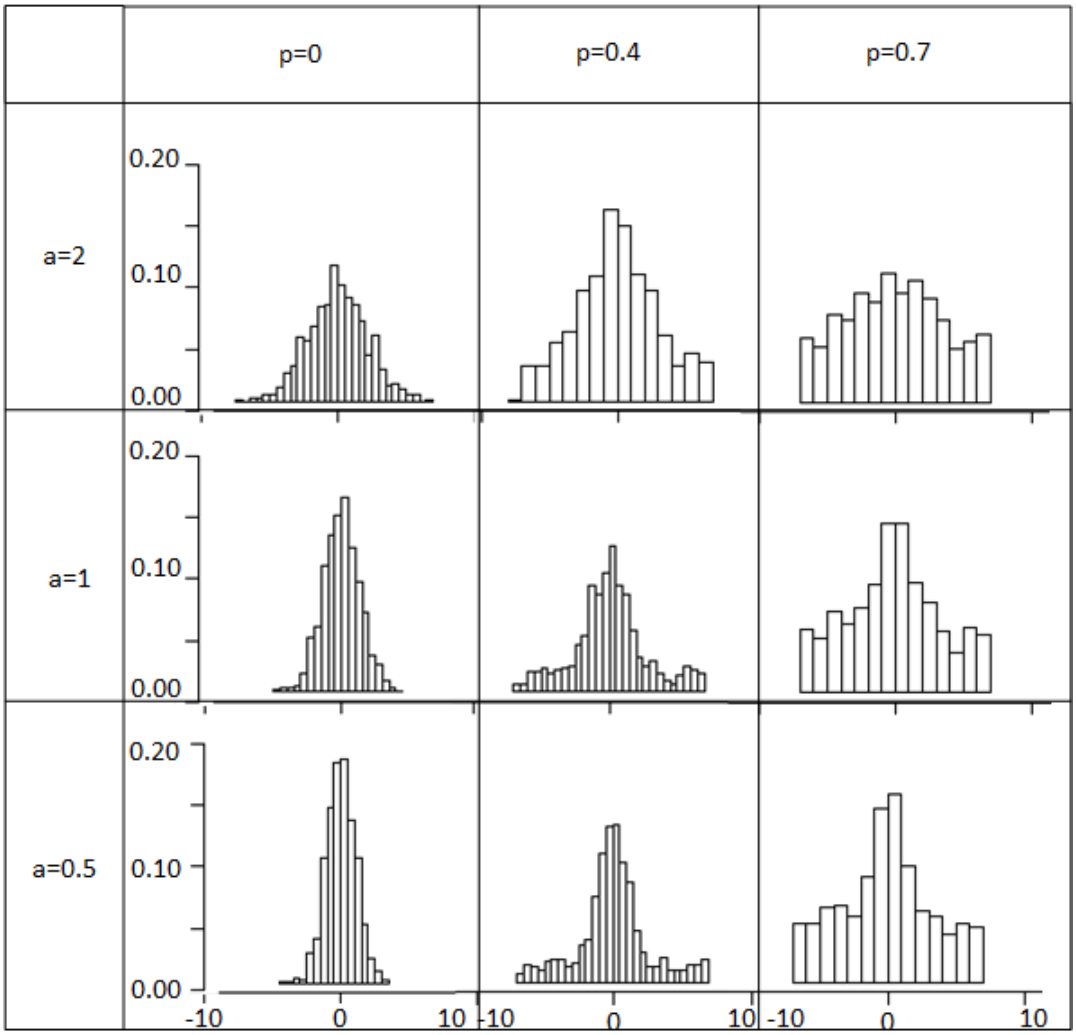

Figure A.10: Analysing the effects of generating a synthetic dataset of 1000 samples (using Equation 5.1) with $\alpha = \{0.5, 1, 2\}$ and $p = \{0, 0.4, 0.7\}$. Note that the y-axis for each histogram is the normalized frequency.

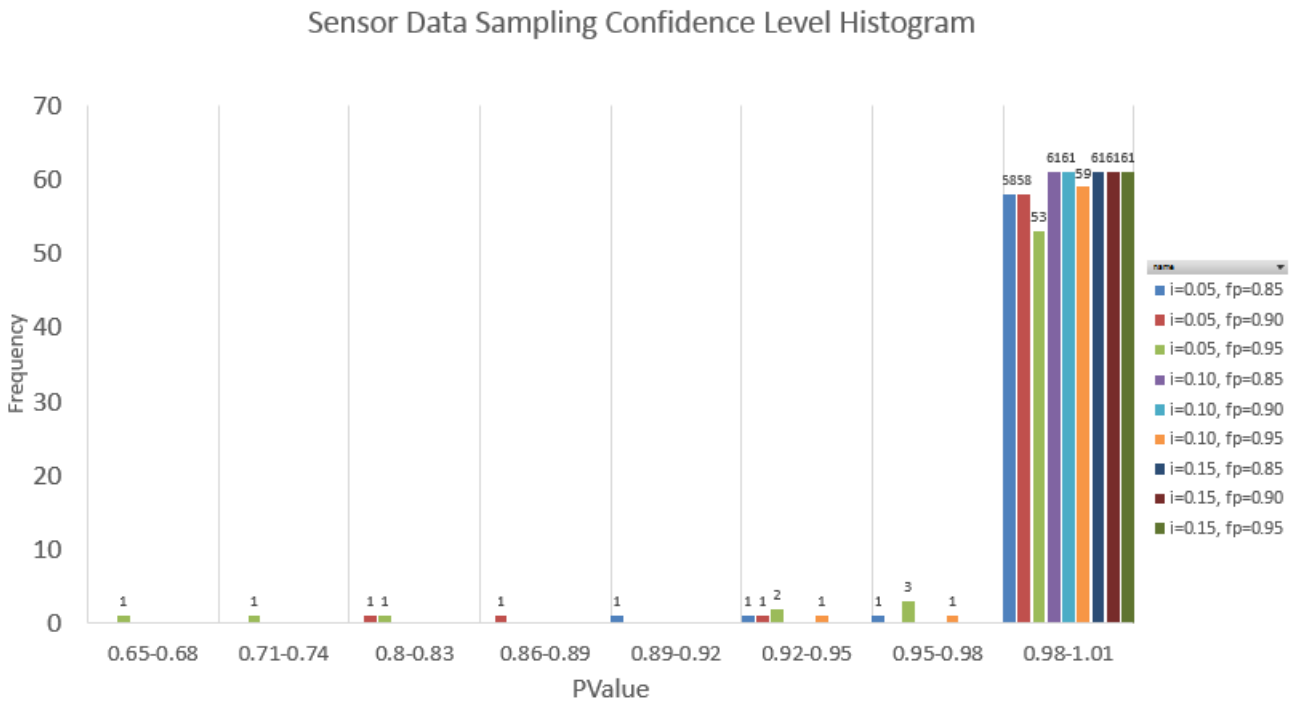

Figure A.11: Confidence levels for sensor data sampling strategies.

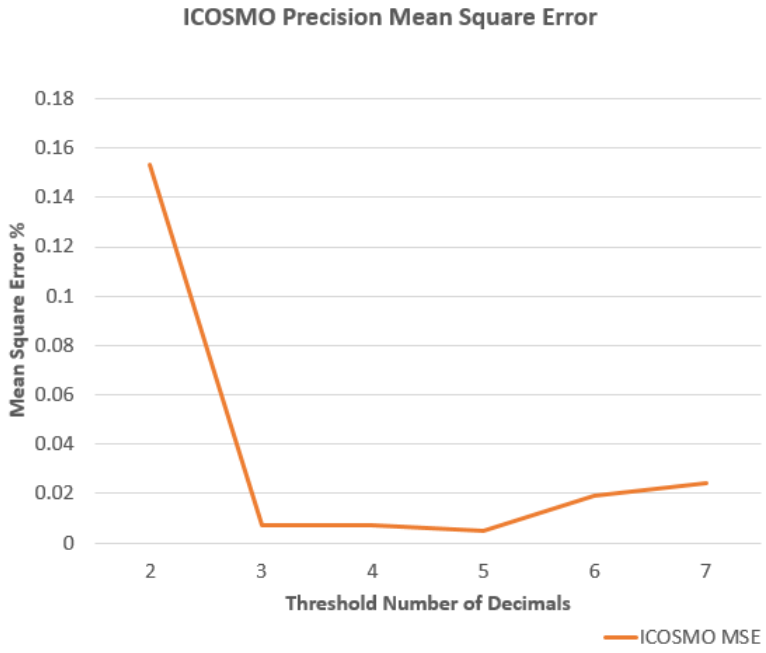

Figure A.12: ICOSMO AUC error when varying threshold decimal precision.

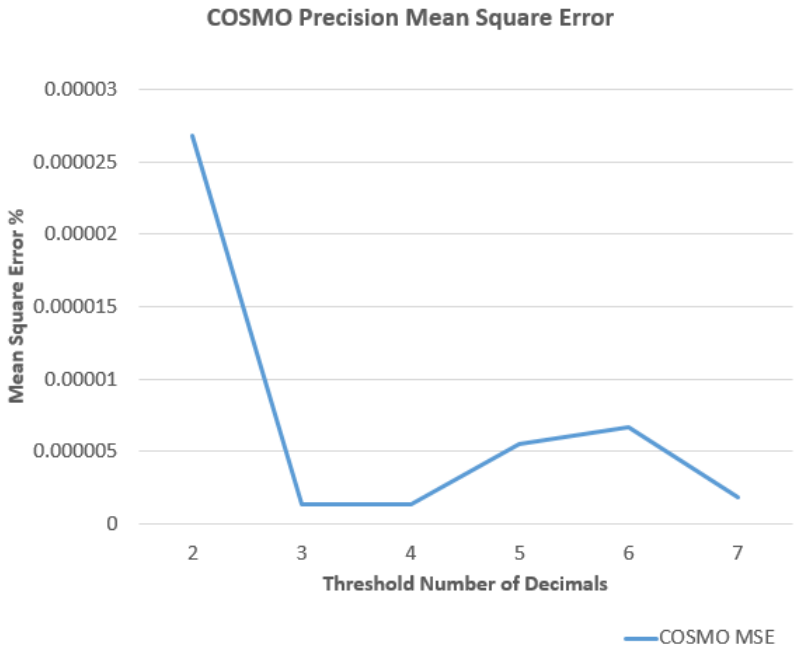

Figure A.13: COSMO AUC error when varying threshold decimal precision.

Appendix B

Configuration and Source Code

B.1 Hardware

This section contains the hardware implementation details of section 5.1.2.

B.1.1 Raspberry Pi

I used Raspberry Pi 3 with Armv7 CPU architecture as the central/master processing board for the gateway. It had Linux (Raspbian) installed, which was used to control all the other boards.

B.1.1.1 Hardware

I used the Raspberry Pi to secure the J1939 interface from the fleet-system. To connect to the J1939 network, I connected the Raspberry Pi to the Copperhill ECU over serial USB connection. It has a USB Wi-Fi modem, named *Long Range Wi-Fi USB with Antenna for Raspberry Pi*, made by Adafruit Industries. This USB Wi-Fi modem provided Wi-Fi connection to the Raspberry Pi. To have a cellular network internet connection, I connected the Raspberry Pi to the Fona 800 via serial connection, using a UART-to-USB bridge adapter. I used the Fona with an activated SIM card to provide the Raspberry Pi with an internet connection via GSM. I also used a UART-to-USB bridge to connect to the Raspberry Pi's serial console via its GPIO pins, to enable debugging via its serial console. I connected the Raspberry Pi to the UPS PIco via GPIO. The PIco lays on top of the Raspberry Pi's pins and manages the power of the Pi via I2C and GPIO. The GPIO pins of interest for this project are those used for accessing the serial console of the Raspberry Pi and pin 21 for controlling the Fona's power (see [76] for pin layout details). For the serial console, pin 2 is used for the 5V power, pin 6 for the ground, pin 8 (GPIO14) for TXD, and pin 10 (GPIO15) for RXD. The Raspberry Pi requires 5V power and the vehicle it was installed on provided 24V, so a DC converter was used.

Name	Description
Raspberry Pi 3	Model B Motherboard. Mini computer that connects everything together
Adafruit Fona 800	GSM modulator, product id 1963
Lithium Ion Polymer Battery - 3.7v 500mAh	Battery for the Fona 800, product id 1578
UPS PIco I2C Control HAT Hardware Version 3.0b	Uninterruptible power supply for the Raspberry Pi
3.7v 450mAh LiPO Battery	UPS PIco's battery
Wi-Fi Antenna	Long range Wi-Fi USB with antenna for the Raspberry Pi by Adafruit Industries
Copperhill ECU	J1939 ECU providing the ability to tap into J1939 network.
Adafruit USB to TTL cable (x2)	UART serial USB bridge, product id 954
Mini USB Type B Female to - Micro USB Male 90 Degree Right Angle Adapter	Used to convert the Mini USB power cable from the DC power converter to Micro USB for the Raspberry Pi
GSM Antenna	Bestbooster Sale 824MHz-2170MHz outdoor antenna and SMA-male connector
Female-USB-to-micro-USB-male	Converter for charging Fona 800 with UPS PIco
DC 24-12V to 5V converter	AUCEE Dash Cam Hardwire kit, vehicle 24-12V to 5V
USB to TTL Cable	Drives UPS PIco power to charge Fona 800's battery
Pin-cables	Female-to-female pin cable from Raspberry Pi's GPIO to Fona 800's key pin
SIM card	Activated SIM card for Fona 800 data plan

Table B.1: Complete list of hardware components used to build the gateway.

Id	Type	Description
1	Boot	Boot up the gateway without Wi-Fi, there should be a Meshcentral connection via GSM.
2	Boot	Boot up the gateway with Wi-Fi, there should be a Meshcentral connection via Wi-Fi.
3	Shutdown	Shutdown the Pi via `'sudo shutdown -h now'` command. Test cases 1 and 2 should hold.
4	Shutdown	Shutdown the Pi via Meshcentral's shutdown command. Test cases 1 and 2 should hold.
5	Shutdown	Shutdown the Pi by cutting the power. Test cases 1 and 2 should hold.
6	Shutdown	Reboot the Pi via Meshcentral's reboot command. Test cases 1 and 2 should hold.
7	Shutdown	Reboot the Pi via `'sudo reboot'` command. Test cases 1 and 2 should hold.
8	Wi-Fi	Turn off the Wi-Fi. A Meshcentral connection should be made shortly after via GSM.
9	Wi-Fi	Long range 30-50ft Wi-Fi should be possible.
10	GSM	Wi-Fi becomes in range when connected to GSM. A Meshcentral connection via Wi-Fi should occur shortly after.
11	Security	Gateway scripts and services should run with non-root privileges.
12	Security	SSH server disabled, which prevents an attack channel.
13	Security	Gateway scripts should have a specific directory they have access to, nothing else.
14	Security	Gateway scripts should not be able to start high privilege services.
15	Security	The Pi should not have any open ports.
16	Security	The Pi's firewall should block all incoming traffic (the NMap tool can be used).
17	Security	Gateway scripts should not be able to communicate directly with the Copperhill ECU.
18	Security	The `'sudo'` command should prompt a password.
19	Power	The gateway should stay powered when power is cut and the power comes back quickly.
20	J1939	The J1939 reading service should start on boot.

Table B.2: STO vehicle node (Raspberry Pi) test cases.

Id	Type	Description
1	Boot	Boot up the SLN with Wi-Fi, there should be a Meshcentral connection.
3	Shutdown	Shutdown the SLN via `'sudo shutdown -h now'` command. Test case 1 should hold.
4	Shutdown	Shutdown the SLN via Meshcentral's shutdown command. Test case 1 should hold.
6	Shutdown	Reboot the SLN via Meshcentral's reboot command. Test case 1 should hold.
7	Shutdown	Reboot the SLN via `'sudo reboot'` command. Test case 1 should hold.
10	Wi-Fi	Wi-Fi becomes in range. A Meshcentral connection should occur shortly after.
11	Security	SLN scripts and services should run with non-root privileges.
12	Security	SSH server disabled, which prevents an attack channel.
13	Security	SLN scripts should have a specific directory they have access to, nothing else.
14	Security	SLN scripts should not be able to start high privilege services.
15	Security	The SLN should not have any open ports other than port 1883 (MQTT's port).
16	Security	The SLN's firewall should block all incoming traffic except for port 1883 (the NMap tool can be used).
18	Security	The `'sudo'` command should prompt a password.
19	Power	The SLN should stay awake when the laptop lid down
20	J1939	J1939 network listening service should start on boot.
21	MQTT	The MQTT broker should start on boot.
21	MQTT	SLN scripts should be able to subscribe to the SLN MQTT broker and receive published messages from the VN

Table B.3: STO server leader node (laptop) test cases.

The Raspberry Pi requires the following installations: Mosquitto for the MQTT client and a Meshcentral agent. Note that the Meshcentral web portal provides the installation script required [65] to configure the agent. The Meshcentral agent ensures the Raspberry Pi stays connected to the administrative server as long as there is internet connection. I added an additional user to ensure services and scripts can run with low privileges. The Wi-Fi antenna should be plugged into one of the Raspberry Pi's USB slots. See section B.2 for detailed Linux commands required to configure the Raspberry Pi for the gateway.

B.1.1.2 Network Interfaces

The Raspberry Pi is responsible for maintaining a persistent internet connection. The only time it has no internet connection is when the vehicle powers down or between Wi-Fi and GSM network interface transitions. The Raspberry Pi prioritizes Wi-Fi, disconnecting from the GSM connection (and turning off the Fona's power) when Wi-Fi becomes available. To turn off the Fona, over GPIO, the Pi will send a signal to the Fona's KEY pin. When the Pi disconnects from Wi-Fi (when it goes out of range, for example), the Raspberry Pi restarts the Fona 800 and establishes a GSM connection. With this network configuration, the gateway can be remotely administered at virtually any point during the vehicle's up-time.

Since the Raspberry Pi is managing a GSM network connection and a Wi-Fi connection, to provide software handles for when the GSM or Wi-Fi connection is established or lost, I chose four directories as the handles. These directories contain scripts that are run, by a gateway networking service, when the following conditions are met respectively: Wi-Fi disconnects, Wi-Fi connects, GSM connects, GSM disconnects. For example, these handles can be used to dump daily J1939 data to a local Wi-Fi-connected SLN. When the Wi-Fi connection is established, by having placed a data dump script in the appropriate directory, the data will be sent to the SLN upon arriving in Wi-fi range.

B.1.2 Fona 800

The Fona 800 can be used to send text messages via SMS, make voice calls, and provide a serially-connected device with an internet connection over GSM. Figure B.1 illustrates a picture of the board. It has a SubMiniature version A (SMA) connector for connecting antennas. The Fona 800 requires an external 3.7V and 500 mAh battery for its GSM modulation, since it will not have enough power when only relying on the Raspberry Pi's 5V input power alone. The battery requires charging via the Fona' Micro USB slot. In fact, the UPS PIco's power output pins provide the necessary power to charge the Fona's battery. By default the Fona 800 is off [1].

I ran an experiment that revealed that when the Fona 800 was on and the Raspberry Pi was on, the Fona's battery charged faster than it drained. In other words, the Fona 800 would therefore always have a charged battery and no additional powering requirements were needed.

The Fona is connected to the Raspberry Pi via its *Key* pin, for toggling power, and via its *Vio*, *RX*, *TX*, and *GND* pins for external power and serial connection with the

Figure B.1: Fona 800 board.

Order	1	2	3	4	5	6	7	8	9	10	11	12
Pin	Bat	GND	+	-	Rst	PS	Key	RI	TX	RX	NS	Vio

Table B.4: Fona 800 bottom breakout pins, top-down order with respect to the orientation of the Fona 800 in figure B.1.

UART-to-TTL cable (see table B.4 for the pin breakout). The serial connection is used to connect the Raspberry Pi to the Fona 800, providing the Raspberry Pi with a GSM internet connection.

B.1.2.1 Pins of Interest

The important pins for the gateway are identified and explained below (full pin descriptions can be found in [1]):

- **Vio** receives 3-5V of input from the Raspberry Pi and powers the Fona 800.
- **RX** Serial UART RX reads data from the Raspberry Pi
- **TX** Serial UART TX writes data to the Raspberry Pi
- **Key** toggles the power of Fona On/Off, by receiving a pulse signal for 2 seconds from the Raspberry Pi's GPIO 21
- **GND** grounds the Raspberry Pi

Section B.2.1 goes into detail about the commands and other configuration required to configure the Fona 800. There are a few libraries required to enable communication between the Raspberry Pi and the Fona. Once they are installed, a few configuration files need to be updated to connect to the GSM network offered by the appropriate service provider.

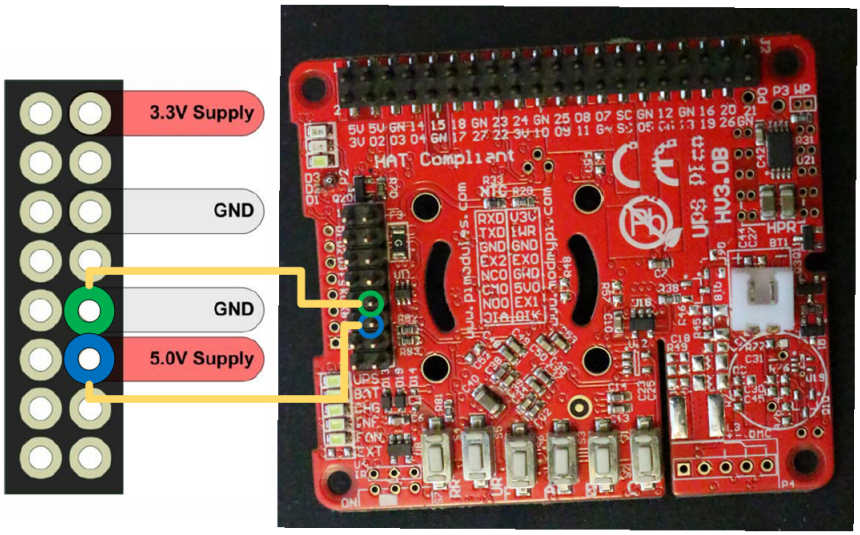

Figure B.2: UPS PIco Board, with pin labeling from [78], where output pins that are used in the gateway are highlighted.

B.1.3 UPS PIco

I used version 3.0 B of the UPS PIco hardware, made by BuyaPi. Figure B.2 illustrates a picture of the board. I used the PIco to manage the power of the Raspberry Pi. When the Raspberry Pi received power, the PIco would power on. When the Raspberry Pi lost power, the PIco would notice and take over by providing power to the Raspberry Pi from its battery, over the GPIO pins. The UPS would last around 30 seconds by default. This time can be configured over I2C. Once the power had been lost for the specified duration, the PIco would send a shutdown signal to the Pi to shut it down gracefully. The PIco charges its battery when power is supplied to it via the Raspberry Pi. The technical user manual of the UPS PIco can be found in [77] and [78].

B.1.3.1 Pins of Interest

Pins of interest on the UPS PIco are the programmable Auxiliary 5V pin (see [78] for more details) and the GND pin. These pins were used to charge the Fona 800's battery over Micro USB, since they meet the Fona 800's battery requirements [1] and they are independent from the Raspberry Pi's power. The charging stops when the Raspberry Pi turns off (30 seconds after the power has been lost). The wiring is done by connecting the PIco's ground and voltage pins to the USB-to-TTL cable, which then is inserted into a USB-female-to-micro-USB cable. The micro USB is inserted in the Fona 800's charging port. For more details on the steps taken to configure the UPS PIco, see section B.2.2.

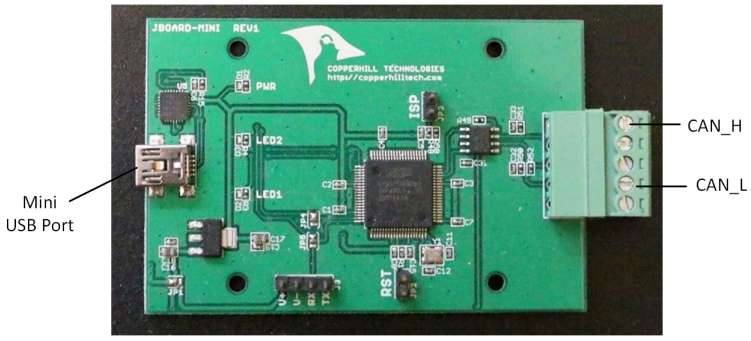

Figure B.3: Copperhill J1939 ECU diagram, where interfaces that are used in the gateway are labeled.

B.1.4 Copperhill ECU

The Copperhill J1939 ECU, illustrated in figure B.3, is used to tap into a J1939 and offer the Raspberry Pi the ability to read from the J1939 network.

B.1.4.1 Simulation a J1939 Network

The ECU J939 can also be used to simulate a J1939 network, which is a useful feature for testing. At least two Copperhill ECUs are required to simulate a J1939 network. That is, one ECU will write J1939 traffic onto the twisted pair cable and the other will read from it. On Windows machines, the *jCOM1939 Monitor* software can be used to connect to a ECU to create J1939 messages.

B.1.4.2 Connection the Copperhill ECU to the Rasbperry Pi

The ECU is connected to the Raspberry Pi via USB (male Mini USB to male standard USB) and connects to the J1939 network using its CAN_H and CAN_L interface via the J1939 twisted pair. A Linux service is implemented on the Raspberry Pi to have read-only access to the STO J1939 network via the Copperhill ECU. The service offers read-only J1939 data to consumers via a Linux pipe. Source code and implementation details for configuring the Raspberry Pi to read from the J1939 network are found in section B.2.3.

B.2 Configuration for the Raspberry Pi for the Gateway

MQTT

MQTT client library installations are required to dump J1939 data to the STO SLN via Linux scripts, which can be done using the command below:

```
sudo apt-get install mosquitto-clients
```

User Configuration

Adding a gateway user, which was accomplished using the command below, is required to make sure that gateway-level services do not run with root privileges.

```
sudo useradd -M <gateway user>
```

Wi-Fi Network Configuration

The Wi-Fi is configured by modifying the file `/etc/network/interfaces`, which is illustrated in the pseudo code below, where `wlan0` indicates the name of the target Wi-Fi interface to enable. Note that the command `ip addr` will reveal all the network interfaces, which is helpful if it is unclear what the name of the target Wi-Fi interface is. The `dhcp` entry indicates the `wlan0` interface will be assigned an IP address from a DHCP server, instead of using a static IP address. The `wpa-ssid` entry is the SSID of the target WPA-enabled Wi-Fi network and `wpa-psk` is the password of the Wi-Fi network. Make sure that there is no `auto eth0` entry in the file, since this will configure the OS to use the `eth0` interface (Ethernet) instead of the Wi-Fi interface.

```
 ...
allow-hotplug wlan0
auto wlan0

iface wlan0 inet dhcp
       wpa-ssid "MyNetworkSSID"
       wpa-psk "mypassword"
...
```

B.2.1 Configuration for the Fona 800

This section goes into the details of the Linux commands required to configure the Fona 800, and to debug it.

Installing Necessary Libraries

To enable the GSM internet connection for the Raspberry Pi via the Fona 800, the following library installations are required:

```
sudo apt-get install ppp screen elinks
```

Debugging Installation

Once the Fona800 is connected to the Raspberry Pi and the required libraries are installed, it can be debugged by accessing its serial console as follows:

Command	Description
`AT`	Echo back to indicate serial command received
`AT+CMEE=2`	Display verbose errors
`AT+CCID`	Display SIM card id
`AT+COPS?`	Display Network Connection
`AT+CSQ`	Display Signal Strength
`AT+CBC`	Display Fona 800 battery remaining life
`CTRL+A`	Typing `:quit` will exit session

Table B.5: Fona 800 useful serial commands for debugging.

```
screen /dev/ttyUSB0 115200
```

where in this case the Fona 800 is device ttyUSB0, and the bit rate is 115200 for the serial connection. To make sure everything has been configured properly, the serial console commands can be used. Table B.5 lists a few key commands, but a full list of commands can be found in [85] and an in depth tutorial can be found in [2] for setting up the Fona 800 with the Raspberry Pi.

Configuring Network Interface

Once the communication with the Fona 800 has been confirmed, the cellular network connection can be configured, that is, the `ppp` network interface. The following commands install the proper configuration files:

```
sudo -i
cd /etc/ppp/peers/
wget https://raw.githubusercontent.com/adafruit/FONA_PPP/master/fona
vi fona
```

, which opens the configuration file in the vi editor. The important line to change is the Access Point Name (APN) value, which tells the Fona what provider to connect to. Replace the "-T ****" entry in the line

```
connect "/usr/sbin/chat -v -f /etc/chatscripts/gprs -T ****"
```

with the proper APN of the SIM card's data plan depending on your service provider. In my case the APN value was `ltemobile.apn` since I used a Fido data plan in this project. It was not clear which APN was appropriate for the data plan I had, so I just tried each possible APN of my region until I found one that allowed me to connect to a cellular network. A description of Canadian APN settings can be found at [6].

To have a persistent network connection on boot, the `/etc/network/interfaces` file needs to be configured to enable the `ppp` network interface on boot, by adding the following to the file:

```
auto fona
iface fona inet ppp
provider fona
```

Care must be taken when configuring the network, since there can only be one default gateway, so the `ppp` interface may not be the default interface used for internet connection.

Powering the Fona

The command below manually powers on the Fona 800's software and configures the `ppp` network interface.

```
sudo pon fona
```

To power off the Fona 800's software and cellular network connection, the command below can be used, but it does not turn off the power of the Fona.

```
sudo poff fona
```

The Python script below toggles the power of the Fona, by sending a low/high signal over GPIO 21 that follows the Fona's Key pin documentation (see [1] for technical documentation). The Python script requires the `RPi.GPIO-0.2.0` python library, and can be found below:

```
from time import sleep
import RPi.GPIO as GPIO
GPIO.setmode(GPIO.BOARD)
GPIO.setup(40, GPIO.OUT)
sleep(20)
GPIO.output(40, False)
sleep(2)
GPIO.output(40, True)
sleep(2)
```

B.2.2 Configuration for the UPS PIco

Installing the UPS Pico onto the Raspberry Pi

To configure the Raspberry Pi and the UPS PIco, make sure the 3.7v 450mAh battery is disconnected from the UPS PIco. Connect the Raspberry Pi's pins to the UPS PIco's pins, such that the UPS PIco is installed over the Raspberry Pi. Once installed, connect the battery to the UPS PIco.

Updating the Raspberry Pi

The Raspberry Pi should then be powered. Update the Raspberry Pi to have the latest packages using the commands below:

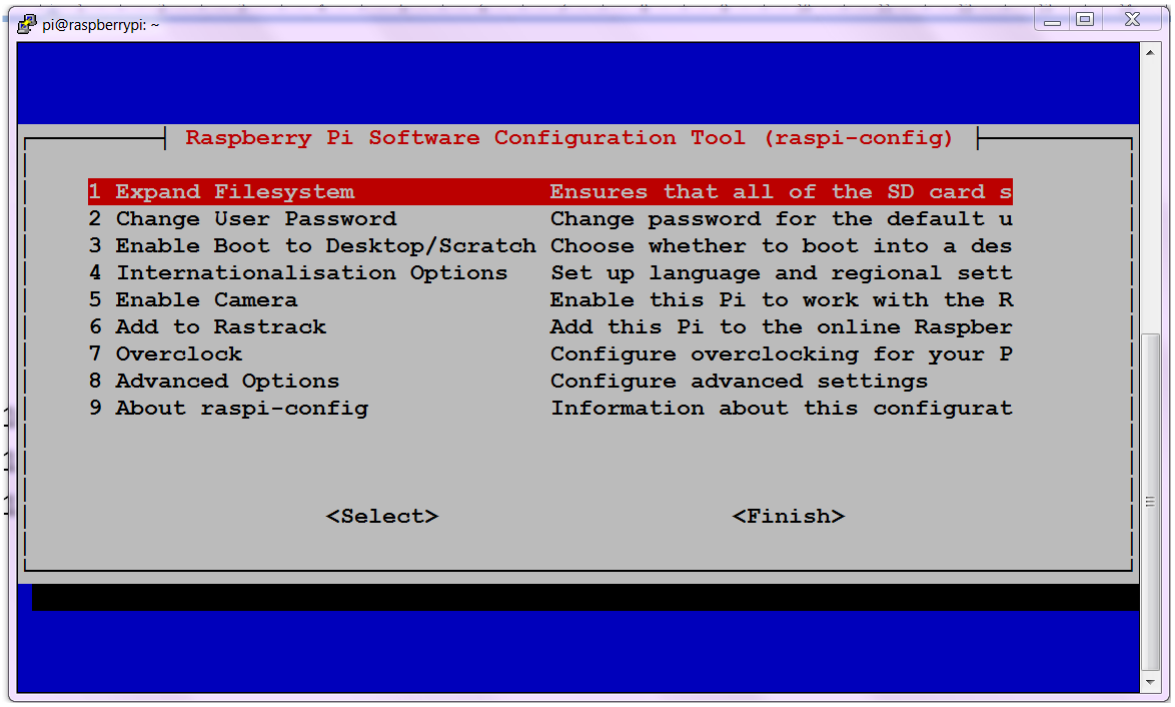

Figure B.4: Raspberry Pi configuration tool console user interface.

```
sudo apt-get update # Fetches the list of available updates
sudo apt-get upgrade # Strictly upgrades the current packages
sudo pat-get dist-upgrade # Installs updates (new ones)
```

Configuring I²C and Serial

I²C must then be enabled to allow the UPS PIco to communicate with the Raspberry Pi, which can be done using the Raspberry Pi configuration tool (see figure B.4), using the command below:

```
sudo raspi-config
```

Select `Advanced Options`, `I2C`, and `Yes` to enable I²C. The serial communication interface must now be enabled. Without exiting the configuration tool, select `Serial`, which may be enabled by default, and select `Yes` when prompted to enable the serial login shell.

Installing Necessary Services and Files

A full tutorial for installing the UPS PIco can be found in [66], describing the next steps required to finish the installation. Once the necessary daemons and files are installed and the UPS PIco's real-time clock is configured, the UPS PIco should then be detected under the Pi's I²C connected devices. To verify this, run

```
sudo i2cdetect -y 1
```

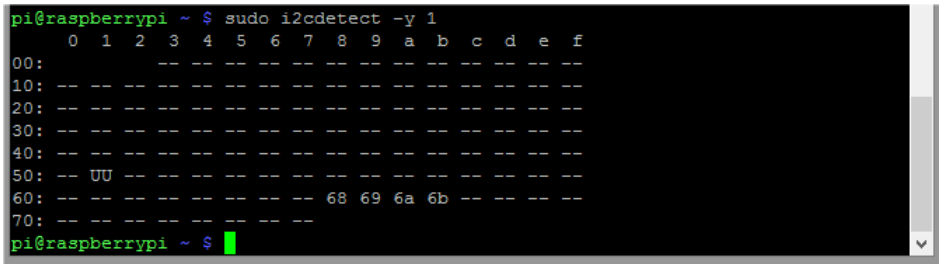

Figure B.5: Example output of the Raspberry Pi's I^{2}C devices.

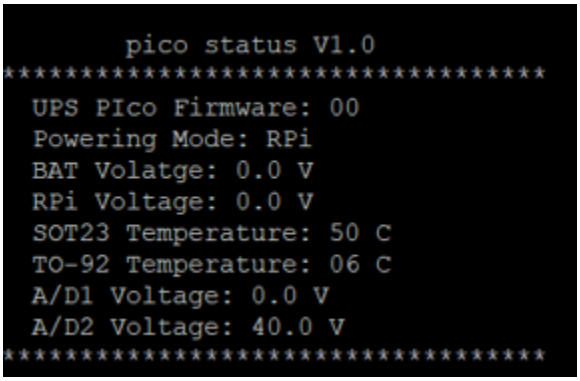

Figure B.6: Example Output of UPS PIco status using `pico_status.py` Python script.

and make sure the entries in the output are not all "`--`". See figure B.5 for example output.

Once the the UPS PIco is confirmed to be properly configured to communicate using I^{2}C with the Raspberry Pi, supporting files must be downloaded to configure the PIco by running the command below:

```
wget http://www.pimodules.com/_zip/UPS_PIco_Supporting_Files.zip
unzip UPS_PIco_Supporting_Files.zip
```

A noteworthy script is the `pico_status.py` script, which outputs the status of the UPS PIco. For example, by running the command below:

```
sudo python pico_status.py
```

the status of the UPS PIco is displayed (see figure B.6 for an example output). Where the `Powering Mode` entry is `RPi` when there is power for the Raspberry Pi, and is `BAT` when there is no power to the Raspberry Pi and the Pi is running off the PIco's battery.

B.2.3 Reading from the J1939 Network Using Copperhill ECU

The C code used are files provided by Copperhill from the detailed Raspberry Pi and Copperhill ECU tutorial in [96]. The C source code does not need to be changed significantly, as only the name of the ECU COM device is machine dependent. So, inside the `config.h` in the source code package, the following entry needs to be changed to reflect the COM devices that represents the Copperhill ECU (in my case its device `ttyUSB0`) for the Raspberry Pi.

```
#define COMPORT                 "/dev/ttyUSB0" /* Serial Port          */
```

Root Privileged J1939 Reading Service

I created a root-privileged service on the Raspberry Pi that communicates over serial UART with the Copperhill ECU. It taps into the STO hybrid test bus J1939 network as read-only. The J1939 reader service loop samples the ECU device at a rate of 1KHz, that is, it checks for a J1939 reading each millisecond.

```
int fd;
int rc;
fd = open("/path/to/pipe",O_WRONLY);

...

while(true)
{

    // Call the COM1939 protocol stack of Copperhill ECU
    int nStatus = COM1939_Receive(&lPGN, &nPriority, &nSourceAddress,
       &nDestAddress, &nData[0], &nDataLen);

    ...

    msg.pgn = lPGN;//copy png into message
    memcpy(msg.data,nData,nDataLen);//copy payload into message

    //publish the msg via fifo pipe
    rc = write(fd,&msg,sizeof(struct J1939Message));


}
```

The service then uses a pipe to output the packets in the following format:

```
struct J1939Message{
   long pgn;
   int size;
   unsigned char data[8];
};
```

Low Privileged J1939 Consumer

Then, whenever lower-privileged gateway scripts need to read from the J1939 network, they need to read from the read-only J1939 pipe. This way, if any vulnerabilities exist, an attacker or a software bug will not be able to write to the STO's J1939 network. Below is C code snippet that illustrates a low-privileged script that is reading from the J1939 pipe.

```
struct J1939Message msg;
int rc;
int fd;
fd = open("/path/to/pipe",O_RDONLY);

while(1){
   rc = read(fd,&msg,sizeof(struct J1939Message));
   ...
}
```

B.3 Configuring the STO Laptop

Updating the OS

The laptop needed to be updated before being deployed to the garage, which was accomplished by running the command below:

```
sudo do-release-upgrade
```

Meshcentral

Once the updates were installed, a Meshcentral agent was installed on the SLN. Installing the Meshcentral agent required the same steps as for the Raspberry Pi (see section B.1.1). A library was required by the Meshcentral agent, which was installed as follows:

```
sudo apt-get install libxtst6
```

Ensure that a script `/etc/init.d/mesh` exists, which will make the Meshcentral agent start on boot and stop on shutdown.

MQTT

The MQTT broker and client must be installed, which is done as follows:

```
sudo apt-get install mosquitto mosquitto-clients
```

Firewall

The firewall needed to be configured to accept incoming MQTT packets (port 1883), so the uncomplicated firewall tool was installed, and the port 1883 was opened, to allow the VN to publish messages to the SLN. This was done using the commands below:

```
sudo apt install ufw #install uncomplicated firewall
sudo ufw enable #enable on boot
sudo ufw default deny incoming #deny the incoming connections
sudo ufw default allow outgoing #simply all outgoing connections, reduces management or
sudo ufw allow 1883/tcp #allow only mqtt traffic through firewalls
```

MySQL Server

I installed My SQL server for data persistence using the command below:

```
sudo apt-get install mysql-server
```

Java

Java was also pre-installed, to avoid installing it via LTE connection at the STO garage, which was done using the command below:

```
sudo apt-get install default-jre
```

Server Leader Node User

An additional user was added, which was done using the command below, to make sure services and scripts can run with low privileges.

```
sudo useradd -M <user>
```

Preventing Laptop From Sleeping

The laptop would go to sleep when the lid was shut, so I configured the laptop to never sleep when the lid is closed as follows:

1. Open the `/etc/systemd/logind.conf` file in a text editor as root

   `sudo vi /etc/systemd/logind.conf`

2. Add a line `HandleLidSwitch=ignore` (make sure it's not commented out)

3. Restart the `systemd` daemon

   `sudo restart systemd-logind` or `sudo service systemd-logind restart`

B.3.1 SLN Data Acquisition Details

The data dumps were sent to the STO SLN using MQTT with the following command on the Raspberry Pi:

```
mosquitto_pub -f /path/to/data-dump-partition -h
    <SLN IP address> -t "/mqtt/datadump/topic"
```

Where `/path/to/data-dump-partition` is the name of the next data dump file partition to send to the SLN. The SLN listened for data dumps by subscribing to the MQTT topic "/mqtt/datadump/topic", and logging all the messages (files) into a file, using the command below:

```
mosquitto_sub -t "/mqtt/datadump/topic" -h localhost > /path/to/outputfile
```

Which would effectively reconstruct all the data dump file partitions since it would append each file's content received to the output file.

B.3.1.1 Difficulties

There were difficulties sending data dumps over Wi-Fi. To acquire data dumps for data analytic purposes, the data had to first partially be dumped to the SLN over MQTT and Wi-Fi when the vehicle would get fueled. Sending huge files was not possible with the resource-constrained Raspberry Pi. It failed to upload 32MB in 10 min, as it was all or nothing. There were a few fuel pumps by the Wi-Fi, while others were over 200 feet away. The Huawei stick only supports a distance of approximately 40m. Upon further investigation, the Wi-Fi connection duration was only solid every third connection, most likely because the best Wi-Fi connection was by the fuel pump nearest to the Huawei stick. As a result, the strategy taken was to fragment the J1939 dump files, and send the compressed version of all the fragments. Instead of sending an entire dump file and hoping the Wi-Fi connection would last long enough to upload, sending file fragments was more consistent and efficient. The data dump files were split into 15KB fragments by using the command

```
split --bytes=15K /path/to/datadump.csv
        /path/to/data-dump-partitions/data-dump-part-
```

Where `/path/to/datadump.csv` is the data dump file, `--bytes=15K` indicates that each partition will be 15 KB in size, `/path/to/data-dump-partitions/` is the output directory to store all the 15 KB file partitions, and each file will be named will the following format `data-dump-part-i`, where i is the patition id.

Field	Possible String Format	Meaning
`PGN`	x	Parameter group number is x
`Parameter Group Label`	s	s is a string
`SPN`	x	Suspect parameter number is x
`SPN Name`	s	s is a string
`Units`	u	u is a string

Table B.6: J1939 Specification Document, consistent sensor definition field formats.

B.4 J1939 and Java

B.4.1 Parsing J1939 Specification Document using Java

This section contains the implementation details of section 5.2.2. Since $J\hat{S}D$ is a CSV file, I iterated the file line-by-line. Let l denote the current line being parsed. For each line l, I parsed each field (column) separately, following the possible formats of each field. Java offers the `String [] String.split(String regex)` function, which makes iterating all the field values in a row straight forward, since it returns an array of string tokens as a result of splitting a string. By using "," as the regular expression (*regex*), the tokens/elements found in the resulting array are each field. The Java J1939 parser then iterates all tokens, and uses the index of the token to determine which field is being parsed. For example, see code snippet below which illustrates how to access the `SPN Name` field for a line with variable name `strLine` :

```
static int SPN_NAME_INDEX = 4;
public void parseJSDLine(String strLine){
    String [] tokens = strLine.split(",");
    String spnName = tokens[SPN_NAME_INDEX];
}
```

There are a couple fields that always have a consistent format throughout the document, which are shown in table B.6. For the string fields, such as `Units`, it is sufficient to just parse it using the logic of the code snippet above. However, some fields are either real numbers or integers, which can be parsed using the Java API `double Double.parseDouble(String doubleString)` and `int Integer.parseInt(String intString)`, to parse doubles and integers respectively.

The JSD does not always follow a structured format (there are occasional typos or a variety of inconsistent formats), so care must be taken to parse every field. For example, the JSD may represent 100% as follows: ("100%","100 %","+100%","100.00%", "+100.0%", etc.). This means `int Integer.parseInt(String intString)` cannot blindly be applied for all integer fields that do not have a consistent format.

Field	Possible String Format	Meaning
`SPN Position in PGN`	x	ESV starting index is byte x
`SPN Position in PGN`	x-y	ESV starts from byte x to byte y
`SPN Position in PGN`	$x.y$	ESV starts from yth bit of byte x

Table B.7: J1939 Specification Document, possible sensor definition formats for the `SPN Position in PGN` field. Note, ESV is an abbreviation for encoded sensor value.

B.4.1.1 Parsing "SPN Position in PGN" Field

The `SPN Position in PGN` field describes the starting position of an encoded sensor value in the J1939 payload. In other words, the desired value to parse from this field is the starting bit index of the encoded sensor value, which can range from 0 to 63. Even though J1939 transport protocols support payload lengths that exceed 8 bytes, on the wire the raw packets can only be up to 8 bytes long, and therefore the bit index from 0 to 63 is sufficient. There are three possible formats the `SPN Position in PGN` field can take, which are shown in table B.7. The Java parser must consider all these cases and check for the presence of the "." and "-" characters to decide how it will parse this field. It once again uses the `String.split` method, but the regular expression argument is "." or "-".

It is worth noting that in the case the `SPN Position in PGN` field has the "x-y" format, the y is not required to parse the starting position. In other words, only parsing x is required for determining the starting bit index for cases "x-y", and "x". In those cases, the resulting starting bit index is $8(x-1)$, where 8 is the number of bits in a byte, and 1 is subtracted to convert indexes from 1-8 to 0-7. In the "$x.y$" case, the result is $8(x-1) + (y-1)$, where we convert x to a bit index before adding y. Note that x, starting from 1, has a range which depends on the length of the payload, and y, starting from 1, has a range from 1 to 8 in the JSD. Since they both start from 1, subtracting 1 converts it to an index. Also, since x starts from 1 and not 0, care must be taken when manipulating arrays with x, since array indices start at 0.

Parsing "SPN Position in PGN" Field

```
public int parseSPNStartingBitPosition(String spnPos){
    int x,y;
    //...

    //case 'x-y' or 'x.y'?
    if(splitIndex != -1){
        String xStr = spnPos.substring(0,splitIndex);
        String yStr = spnPos.substring(splitIndex+1, spnPos.length());
        x = Integer.parseInt(xStr) -1;
        y = Integer.parseInt(yStr) -1;
    }else{//case 'x'
        x = Integer.parseInt(spnPos);
    }
    x = x*8;
```

Field	Possible String Format	Meaning
`SPN Length`	x bits	ESV length is x bits
`SPN Length`	1 bit	ESV value is 1 bit
`SPN Length`	x bytes	ESV length is x bytes
`SPN Length`	1 byte	ESV length is 1 byte

Table B.8: J1939 Specification Document, possible sensor definition formats for the `SPN Length` field.

```
        //note, bitCase is true when 'x.y' is format
        return bitCase ? x + y: x;

    }
```

B.4.1.2 Parsing "SPN Length" Field

This field either defines a length in bits or a length in bytes (see Table B.8). Since the encoded sensor values can be only a couple of bits in length, it is simpler to just manipulate everything in number of bits. With the help of `String String.split(String regex)`, using white space as the regular expression argument, the 1st token, x, in the resulting token array is the desired length. The 2nd token is used to determine if x is interpreted as a bit length or byte length.

Parsing "SPN Length" Field

```
    public int parseSPNLength(String spnLen){

        String [] tokens = spnLen.split("\\s+");//"\\s+" is regex for many
            whitespaces
        int x = Integer.parseInt(tokens[0]);

        //working with byte lenght instead of bits?
        if(spnLen.contains("byte")){
            x = 8*x;
        }
        return x;

    }
```

B.4.1.3 Parsing "Resolution" Field

The `Resolution` field is very unstructured. Most of the entries follow the format of the resolution followed by unnecessary information, for example "1/256 rpm" or "-40 °C"

Field	Possible String Format	Meaning
`Resolution`	2^x states/x bit	resolution is 1 since ECV is a binary number
`Resolution`	Binary	resolution is 1 since ECV is a binary number
`Resolution`	x u per bit	resolution is x, for some unit u (a string)
`Resolution`	x u/bit	resolution is x, for some unit u (a string)
`Resolution`	$1/x$ u per bit	resolution is x, for some unit u (a string)
`Resolution`	$1/x$ u/bit	resolution is x, for some unit u (a string)
`Resolution`	bit-mapped	It appears to be a binary status or command
`Resolution`	ASCII	no resolution, the ESV is a string
`Resolution`	x u	resolution is x of u units (a string)

Table B.9: J1939 Specification Document, possible sensor definition formats for the `Resolution` field.

(see Table B.9). Therefore, most of these formats require only parsing the 1st element of the string to obtain the desired resolution. However, there are a few special cases. For example, sensors with "bit" as their unit all follow the following format: " 2^x states/x " where x is the length in bits of the encoded sensor value. The resolution for this type of sensor is simply 1, since its a binary number and the encoded value can also be interpreted as already being decoded. For purposes of simulation simplicity, the sensor definitions with the following units were just ignored when parsing the JSD: "Binary", "ASCII", and "bit-mapped".

Parsing "Resolution" Field

```
public double parseResolution(String res){
    //... skipping unwanted resolutions

    if(res.contains("states")) return 1.0;
    String [] tokens = res.split("\\s+");//"\\s+" is regex for many
       whitespaces

    //case '1/x units' instead 'x units'?
    if(res.startsWith("1/")){

        String divisionStr = tokens[0];
        //remove 1/ from string
        divisorStr = division.substring(2,resTmp.length());
     double divisor = Double.parseDouble(divisorStr);
     return 1.0/divisor;
  }else{
     return Double.parseDouble(tokens[0]);
  }
}
```

Field	Possible String Format	Meaning
`Offset`	x	offset is x
`Offset`	x u	offset is x of u units (a string)

Table B.10: J1939 Specification Document, possible sensor definition formats for the `Offset` field.

Field	Possible String Format	Meaning
`Data Range`	x to y	x is the minimum, y is the maximum
`Data Range`	x to y u	x is the minimum, y is the maximum of u units (a string)

Table B.11: J1939 Specification Document, possible sensor definition formats for the `Data Range` field.

B.4.1.4 Parsing "Offset" Field

There are not many different formats (see table B.10) for this field, and it is very similar to parsing the resolution.

B.4.1.5 Parsing "Data Range" Field

This field is also straightforward to parse, since there are not many different formats (see table B.11). To parse the `Data Range` field, the tokens are split by the regular expression *"to"*. As a result, the first element of the split is the theoretical minimum value of the sensor, and the second element of the split is the maximum. It is also worth mentioning that the minimum value the sensor can have is also its `Offset` field, since the smallest possible encoded value is 0.

Parsing "Data Range" Field

```
public void parseDataRange(String dataRange){

    String [] tokens = dataRange.split("to");
    double min = Double.parseDouble(tokens[0]);
    double max = Double.parseDouble(tokens[1]);

    //...
}
```

B.4.1.6 Parsing "Operational Range" Field

The operational range field is a unique case (see table B.12). For most sensor definitions in the JSD, they do not have an entry for this field. It describes the expected behavior of a sensor. In total there were 18 375 JSD entries, and 189 entries have operational ranges.

Field	Possible String Format	Meaning
`Operational Range`	$\emptyset$	some sensors do not have an operational range
`Operational Range`	x to y	*
`Operational Range`	x to y u	**
`Operational Range`	x to yu	**
`Operational Range`	x to $+y$ u	**
`Operational Range`	x u to y u	**
`Operational Range`	x u to y	**
`Operational Range`	xu to yu	**
`Operational Range`	x u dir_1 to y u dir_2	** and dir_i is some direction
`Operational Range`	*verbose*	Unformatted text

Table B.12: J1939 Specification Document, possible sensor definition formats for the `Operational Range` field. (*) denotes: x is the minimum and y is the maximum operational range. (**) denotes x is the minimum and y is the maximum operational range, where u is a string representing the units.

Parsing "Operational Range" Field

```
public void parseOperationRange(String opRange){
    //replace symboles ('"','+',',', and '%') that will make parsign
       difficult
    String token = opRange.replaceAll("\"","").replaceAll(",","").
    replaceAll("%","").replaceAll("\\+","");

  //split into two peices
  String [] opRangeTokens = token.split("to");

  double minOpRange = parseOperationRangeDouble(opRangeTokens[0]);
  double maxOpRange = parseOperationRangeDouble(opRangeTokens[1]);
}

public double parseOperationRangeDouble(String s){
  String[] tokens = s.split("\\s+");//split by white space

  return Double.parseDouble(tokens[0]);
}
```

B.4.2 J1939 Packet Analysis using Java

This section contains the implementation details of section 5.2.3. The Java J1939 decoding program iterates the data line by line. Each line contains a packet. For each J1939 packet

(j), each sensor (SPN) s in the parameter group p may have an encoded reading in the packet's payload. Let κ denote the set of sensor definitions obtained from parsing the $J\hat{S}D$. For some PGN (n), let κ_n denote the set of sensor descriptions of the SPNs (or sensors) of n found in the JSD. The payload bytes found in the CSV packets in the data dump file were processed. For each byte, represented as a hex string, they were converted to an actual byte in Java using `Integer.parseInt(hexByteStr, 16)`, where 16 indicates the string should be interpreted as a base 16 (hexadecimal) number. For each packet, a byte array of length 8 was created and populated. Once the payload was converted to a byte array, it was time to extract the encoded sensor values.

B.4.2.1 Extracting Encoded Sensor Values

The first step is to extract encoded sensor values from the payload of j. As the packets are being iterated, for each s in κ_p, extract the encoded value, e of s, from the payload of j. There are two JSD fields that are required to extract e from j, "`SPN Position in PGN`" and "`SPN Length`"(see section B.4.1 for more details on these fields). Both these fields are used to find which subset of bits in j's payload represents e. Let i be the starting bit position of e (s's "`SPN Position in PGN`"), and let j be the length (number of bits) of e (s's "`SPN Length`"). Let P be the payload of j and $P_{x,y}$ denote a binary word, a set of bits, from the xth bit to the yth bit (inclusively) of P. For example, since the J1939 packets of the data dump have payloads of 8 bytes, then $P_{0,63}$ would be the entire payload, $P_{0,7}$ would be the 1st byte, and $P_{8,39}$ would be the 4-byte word starting from the 2nd byte. So in other words, $e = P_{i,i+j-1}$. See figure B.7 for an example of extracting $P_{15,26}$. I implemented this in Java by using the `java.nio.ByteBuffer` class. It provides simple bit manipulation operations; for example, converting a `long` variable into a byte array. Once the payload is converted to a `long` variable v, to extract e, 2 bit shift operations are required on v. First, a shift by i bits to the left is performed, and second, a shift by $64 - j$ bits to the right is performed. This results in $e = v$. This type of operation is sensitive to the Endianness of the machine. The machine used to decode the values was Little Endian byte ordering, and since J1939 is also Little Endian, I did not need to take any Endian conversion measures.

B.4.2.2 Decoding Sensor Values

Once the encoded sensor values were extracted, it was time to decode them. For sensors that have the "bit" unit of measurement, there was nothing to do, as the encoded value was the sensor value. For sensors that have a unit of measurement other than "bit", simply applying the equation $y = mx + b$ to the encoded sensor value x will decode the value, where for some sensor s, m is the resolution of s, b is the offset of s, and y is the decoded value. The following is an example of decoding the J1939 packet below:

```
380,61450,3e,69,1f,30,10,0,59,ff
```

The PGN 61450 refers to the parameter group "Engine Gas Flow Rate". Suppose one wants to obtain the sensor value of the SPN 132, which is the "Engine Intake Air Mass

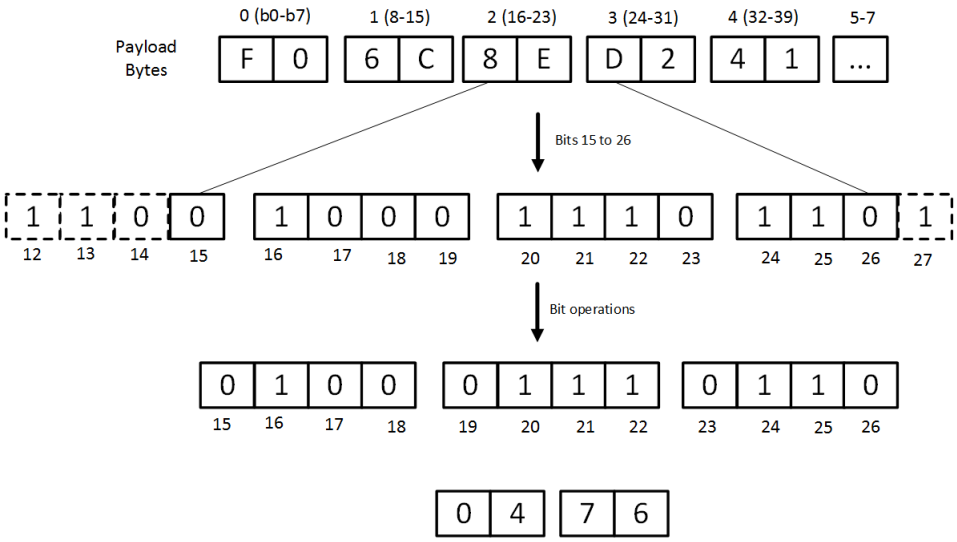

Figure B.7: Example of extracting a subset of bits from a payload.

Flow Rate" parameter. The encoded value of this parameter starts at the 3rd byte, and it has a length of 2 bytes, resolution of 0.05 kg/h per bit, and offset of 0 kg/h. The steps below illustrate the sensor value decoding:

$$
\begin{aligned}
&\texttt{3e 69 } \mathbf{\texttt{1f 30}} \texttt{ 10 0 59 ff} \\
&b = 0 kg/h \\
&m = 0.05 \texttt{(kg/h)/bit} \\
&x = \texttt{1f 30} = (\texttt{1f30})_{16} = (7984)_{10} \texttt{ bit - in Little Endian systems} \\
&x = \texttt{1f 30} = (\texttt{301f})_{16} = (7984)_{10} \texttt{ bit - in Big Endian systems} \\
&y = mx + b \\
&y = (0.05 \texttt{ (kg/h)/bit}) \cdot (7984 \texttt{ bit}) + 0 \texttt{ kg/h} \\
&y = 0.05 \cdot 7984 \texttt{ kg/h} + 0 \texttt{ kg/h} \\
&y = 0.05 \cdot 7984 \texttt{ kg/h} + 0 \texttt{ kg/h} \\
&y = \mathbf{399.2} \texttt{ kg/h}
\end{aligned}
\tag{B.1}
$$

The decoded value is 399.2 kg/h.

B.5 Fleet Simulation Java Code

B.5.1 Noise Generation

Below are the implementation details for generating random noise of section 5.3.4:

```
public double generateNoise(){
```

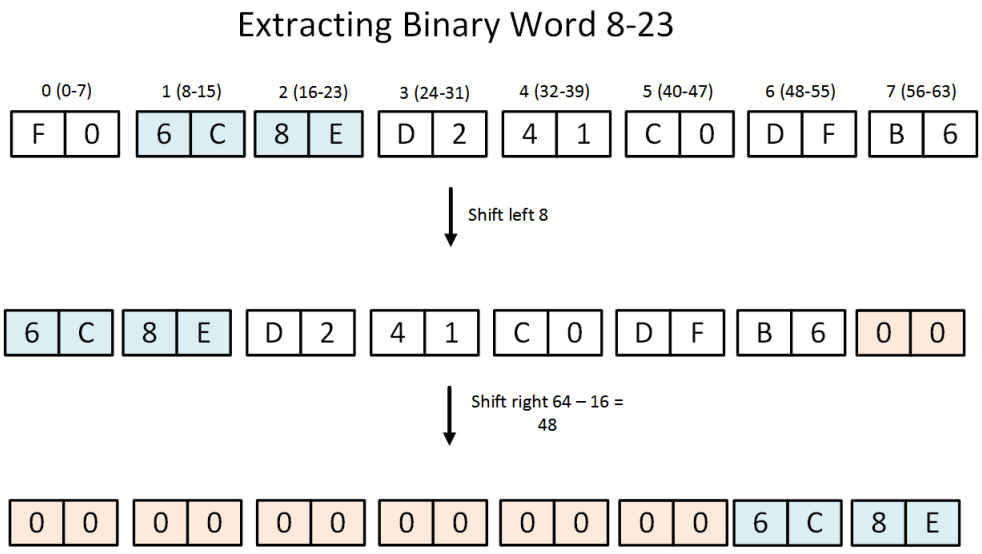

Figure B.8: Bit operations required to extract bits 16 to 24, a subset of bits from a payload.

```
    if (type == Distribution.NORMAL){
        return rNorm.nextDouble();
    }else{
        return rUni.nextDouble();
    }
}
```

B.5.2 Configuration Phase

This section contains the implementation details of section 5.3.5.

B.5.2.1 Configuration File XML Format

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
  <comment>a comment for the configuration file</comment>
  <entry key="key">value</entry>
</properties>
```

B.5.2.2 Reading Configuration File

```
    Properties properties = new Properties();
    properties.loadFromXML(new FileInputStream(new File(filePath)));
    String value = properties.getProperty("target.key");
```

B.5.2.3 Configuration File - Class Room Object

```
<entry key="id">class-room-123</entry>
<entry key="student0">1234</entry>
<entry key="student1">1235</entry>
...
<entry key="studentn">123n</entry>
```

B.5.2.4 Configuration File - School Object

```
<entry key="class-room-path0">/home/school/class-room-0.xml</entry>
<entry key="class-room-path1">/home/school/class-room-1.xml</entry>
...
<entry key="class-room-pathn">/home/school/class-room-n.xml</entry>
```

B.5.3 Generation Phase - COSMO

This section contains the implementation details of section 5.3.6.3.

B.5.3.1 SensorMap Class Members

```
public class SensorMap{

   private HashMap<Algorithm,
                   HashMap<Vehicle,
                   HashMap<Sensor,SensorInstance>>> sensorInstanceMap;
   private HashMap<Algorithm,
                   HashMap<Sensor,
                   List<SensorInstance>> sensorClassMap;

}
```

B.5.3.2 Sensor Map Comparison

```
    M.sensorInstanceMap.get(a).get(v).get(s) ==
       M.sensorClassMap.get(a).get(s).get(0);
```

B.5.3.3 Histogram Computing Bin Index

```
    int computeBinIndex(double sample) {
        if(sample > max) return bins.length-1;
        if(sample < min) return 0;

       return (int) Math.floor(sample/this.binWidth);
   }
```

B.5.4 Analysis Phase

This section contains the implementation details of section 5.3.7.

B.5.4.1 ROC Curve Point Output

```
public class ROCCurvePoint extends Event{

   private Algorithm algorithm;
   private int truePositiveCount;
   private int trueNegativeCount;
   private int falsePositiveCount;
   private int falseNegativeCount;
   private double varyingThreshold;
   private double falsePositiveRate;
   private double truePositiveRate;
   private double accuracy;
   private double fscore;
}
```

B.5.4.2 Decimal Manipulation

```
Double truncatedThreshold = BigDecimal.valueOf(threshold)
                            .setScale(precision, RoundingMode.HALF_UP)
```

```
.doubleValue();
```

B.5.5 Analysis Phase - COSMO

This section contains the implementation details of section 5.3.7.1.

B.5.5.1 Counting Detection Rates

```
    if(faultHistory.isSensorFaultInvolved(...)){
        if(deviationOccured){
         tp++;
        }else{
         fn++;
        }
    }else{
        if(deviationOccured){
         fp++;
        }else{
         tn++;
        }
    }

}
```

B.5.6 Analysis Phase - Output

This section contains the implementation details of section 5.3.7.4.

B.5.6.1 Writing a History Event

```
public class HistoryEvent extends Event implements Serializable{
   private SensorStatusHistory sensorStatusHistory;
   private FaultHistory faultHistory;
   private RepairHistory repairHistory;
}

public void writeHistoryEvent(String outputFile,HistoryEvent histEvent){
    FileOutputStream fileOut =
            new FileOutputStream(outputFile);
```

```
    ObjectOutputStream out = new ObjectOutputStream(fileOut);
    out.writeObject(histEvent);
}
```

B.6 Creating ROC Curves Using the R Programming Language

Load the CSV file into memory as follows:

```
outputData = read.csv(file="/path/to/roc-curve.csv",
                  header=TRUE,
                  sep=",")
colnames(outputData)[i] <- "column i's name"
```

Creating the ROC Curve

Once the CSV is read, the data is sorted by threshold values smallest to biggest, since ROC curves are plotted from left to right starting from threshold 0 and ending at threshold 1.

```
outputData = outputData[order(outputData$threshold),]
```

Since each algorithm has its own curve, each algorithm will be assigned a color, using the algorithms' id. R factors can be used on text data to allow the text data to be used as indexes. This is useful, since ICOSMO algorithms are differentiated by their respective COSMO algorithm by adding 64 to the id of the COSMO algorithm. This means that there is a big gap between COSMO algorithm's id and ICOSMO algorithms' id. So, making them into factors will let the ids be treated as consecutive integers to be used as indices. This way, the id of the algorithms will be used to choose what color to graph them as, which is defined in a color array.

```
mycolors =c(...)
outputData$algorithm_id = factor(outputData$algorithm_id )
```

For example, let `outputData$algorithm_id` be equal `c(0,0,64,1,63,1,64,63)`, where c is a function in R that creates a vector, then `mycolors[outputData$algorithm_id] = c('green','green','yellow','red','black','red','yellow','black')`. A similar process can be done for choosing the type of symbol that will be used to represent an algorithm's point on the ROC curve.

To plot the ROC curve and save it to the file $f_{\texttt{ROC}}$, the `plot` function is used

```
    #specify output file
    png(file = output_roc_curve_file)
```

```
   #plot roc curve
   plot(rocCurvePoints$fpr, #x axis false positive rate
        rocCurvePoints$tpr,#y axis true positive rate
        type = "p", #plot points
        pch=mysymboles[rocCurvePoints$algorithm_id], #symboles for each
           alg's point
        xlim=c(0,1), # x axis limited from 0 to 1
        xlab="False Positive Rate", #x axis label
        col=mycolors[rocCurvePoints$algorithm_id], #colors for each alg's
           point
        ylim=c(0,1), #y axis lmited from 0 to 1
        ylab="True Positive Rate", #y axis label
        main="ROC Curve") #chart title
```

For more information on ROC curves see section 2.13.1. Due to time constraints, I did not remove any points from the ROC curves, which is still analytically correct. Instead I plotted all the ROC curve points. Future work could involve analyzing the ROC curve after applying this smoothing. Once the ROC curves are rendered, the area under each of them is computed. To compute the area under the curve, the R package `MESS`, Miscellaneous Esoteric Statistical Scripts version 0.5.5, is used. For each algorithm, the area under the curve is computed. The FPR and TPR are already ordered, so no further ordering is required for the `MESS` package's `auc` function. It performs linear interpolation between points.

```
library("MESS")
for(algId in unique(outputData$algorithm_id)){

   #get only data of current algorithm
   algData = outputData[outputData$algorithm_id == algId,]

    area = auc(algData$fpr,
               algData$tpr,
               from = min(algData$fpr),#start from smallest value, ie 0
               to = max(algData$fpr),#end at the biggest value, ie 1
               type = c("linear"),#linear interpolation
               absolutearea = FALSE)

    #append area to AUC vector
    areasUnderCurveVec = c(areasUnderCurveVec,areaUnderCurve)
    #append algorith id to id Vector
    idVec = c(idVec,algId)
    ...
}
```

The AUC for each algorithm is saved into a R data frame. The data frame has 2

columns, *algorithm_id*, and *auc*. The data frame is saved to the file $f_{\texttt{AUC}}$, using the following R code

```
 outputDataFrame = data.frame(idVec,areasUnderCurveVec);
colnames(outputDataFrame)[1] <- "algorithm_id"
colnames(outputDataFrame)[2] <- "auc"
write.csv(outputDataFrame, file = output_auc_csv,row.names=FALSE,
   na="")
```

The final step is to create the bar chart that compares the AUC of all the histogram distances used in COSMO and ICOSMO. The following R code is used to create the bar chart and save it to $f_{\texttt{BC}}$, where `lookup` is a data frame with the following columns:

1. *algorithm_id*: id of the algorithm
2. *algorithm_name*: name of histograms distance, with "-I" appended to it for ICOSMO
3. *algorithm_raw_name*: simply name of histograms distance
4. *type*: ICOSMO or COSMO

```
barplot(lookup$auc, #area under curve is used as value for bars
  main="Histogram Distance Area Under ROC Curve Comparison", #chart
     title
  horiz=TRUE, #bars are displayed horizontally
  xlim=c(0,1.1),# x axis is from 0 to 1.1 for a bit of space
  ylab="Algorithm", #y axis label
  xlab = "Area Under ROC Curve", #x axis label
  col = mycolors[lookup$type], #colors of the bar flunctate between 2
     colors depending on type of algorithm
  names.arg = lookup$algorithm_raw_name, # bar labels will be the raw
     name (HELLINGNER, HELLIGNER-I)
  las=1,#this makes the x-axis labels normal, and y-axis labels side
     ways algorithm names 90 degrees, and y axis 90 degress
  space = rep(c(1,0),length(lookup$algorithm_id)/2), #makes space
     beteen every 2 bars, to make groups
  cex.names = 0.5)
```

References

[1] L. Ada. Adafruit learning system, adafruit fona. [Online]. Available: https://cdn-learn.adafruit.com/downloads/pdf/adafruit-fona-mini-gsm-gprs-cellular-phone-module.pdf

[2] ——. Overview. [Online]. Available: https://learn.adafruit.com/adafruit-fona-mini-gsm-gprs-cellular-phone-module?view=all

[3] M. Amarasinghe, S. Kottegoda, A. L. Arachchi, S. Muramudalige, H. M. N. D. Bandara, and A. Azeez, "Cloud-based driver monitoring and vehicle diagnostic with obd2 telematics," *2015 Fifteenth International Conference on Advances in ICT for Emerging Regions (ICTer)*, 2015.

[4] Anybus. (2018) Industrial internet of things and embedded devices hardware manufacturer. [Online]. Available: https://www.anybus.com/

[5] S. S. Bandyopadhyay, A. K. Halder, P. Chatterjee, M. Nasipuri, and S. Basu, "Hdk-means: Hadoop based parallel k-means clustering for big data," in *Calcutta Conference (CALCON), 2017 IEEE.* IEEE, 2017, pp. 452–456.

[6] R. Behar. (2017) How to use apn settings for canadian carriers. [Online]. Available: https://mobilesyrup.com/2017/03/03/apn-settings-canadian-carriers-telus-bell-rogers-freedom/

[7] C. Bell, *MySQL for the internet of things.* Apress, 2016.

[8] L. Belli, S. Cirani, G. Ferrari, L. Melegari, and M. Picone, "A graph-based cloud architecture for big stream real-time applications in the internet of things," in *European Conference on Service-Oriented and Cloud Computing.* Springer, 2014, pp. 91–105.

[9] L. G. Birta and G. Arbez, *Modelling and simulation.* Springer, 2013.

[10] F. Bonomi, R. Milito, P. Natarajan, and J. Zhu, *Big data and internet of things: A roadmap for smart environments*, 2014, vol. 546.

[11] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Proceedings of the first edition of the MCC workshop on Mobile cloud computing.* ACM, 2012, pp. 13–16.

[12] A. Boukhdhir, O. Lachiheb, and M. S. Gouider, "An improved mapreduce design of kmeans for clustering very large datasets," in *Computer Systems and Applications (AICCSA), 2015 IEEE/ACS 12th International Conference of*. IEEE, 2015, pp. 1–6.

[13] D. R. Braden and D. M. Harvey, "A prognostic and data fusion based approach to validating automotive electronics," *SAE Technical Paper Series*, Jan 2014.

[14] A. P. Bradley, "The use of the area under the roc curve in the evaluation of machine learning algorithms," *Pattern recognition*, vol. 30, no. 7, pp. 1145–1159, 1997.

[15] J. Budakoti, "An iot gateway middleware for interoperability in sdn managed internet of things," Ph.D. dissertation, Carleton University, 2018.

[16] S. Butylin, "Predictive maintenance framework for a vehicular iot gateway node using active database rules," Master's thesis, Université d'Ottawa/University of Ottawa, 2018.

[17] S. Byttner, T. Rgnvaldsson, and M. Svensson, "Consensus self-organized models for fault detection (cosmo)," *Engineering Applications of Artificial Intelligence*, vol. 24, no. 5, pp. 833 – 839, 2011. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0952197611000467

[18] S. Byttner, S. Nowaczyk, R. Prytz, and T. Rognvaldsson, "A field test with self-organized modeling for knowledge discovery in a fleet of city buses," *2013 IEEE International Conference on Mechatronics and Automation*, 2013.

[19] S.-H. Cha, "Taxonomy of nominal type histogram distance measures," *City*, vol. 1, no. 2, p. 1, 2008.

[20] T. Chou, *Precision - Principles, Practices and Solutions for the Internet of Things:*. McGraw Hill Education, 2017. [Online]. Available: https://books.google.ca/books?id=OVpHDwAAQBAJ

[21] S. K. Datta, C. Bonnet, and N. Nikaein, "An iot gateway centric architecture to provide novel m2m services," in *Internet of Things (WF-IoT), 2014 IEEE World Forum on*. IEEE, 2014, pp. 514–519.

[22] J. Davis and M. Goadrich, "The relationship between precision-recall and roc curves," in *Proceedings of the 23rd international conference on Machine learning*. ACM, 2006, pp. 233–240.

[23] M. Díaz, C. Martín, and B. Rubio, "State-of-the-art, challenges, and open issues in the integration of internet of things and cloud computing," *Journal of Network and Computer Applications*, vol. 67, pp. 99–117, 2016.

[24] B. Ding, D. Mends, I. Kiringa, and T. Yeap, "Iot architecture with hybrid communication modes for fleet management," in *Technical report, Faculty of Engineering*. University of Ottawa, Ottawa, Ontario, Canada.

[25] O. A. Doreswamy and B. Manjunatha, "Scalable k-means algorithm using mapreduce technique for clustering big data."

[26] S. H. Dsilva, "Diagnostics based on the statistical correlation of sensors," *SAE International Journal of Passenger Cars - Electronic and Electrical Systems*, vol. 1, no. 1, p. 5361, 2008.

[27] H. M. Elattar, H. K. Elminir, and A. Riad, "Conception and implementation of a data-driven prognostics algorithm for safety–critical systems," *Soft Computing*, pp. 1–18, 2018.

[28] Y. Fan, S. Nowaczyk, and T. Rgnvaldsson, "Incorporating expert knowledge into a self-organized approach for predicting compressor faults in a city bus fleet," 2015.

[29] Y. Fan, S. Nowaczyk, and T. Rögnvaldsson, "Evaluation of self-organized approach for predicting compressor faults in a city bus fleet," *Procedia Computer Science*, vol. 53, pp. 447–456, 2015.

[30] T. Fawcett, "An introduction to roc analysis," *Pattern recognition letters*, vol. 27, no. 8, pp. 861–874, 2006.

[31] C. Fiandrino, F. Anjomshoa, B. Kantarci, D. Kliazovich, P. Bouvry, and J. N. Matthews, "Sociability-driven framework for data acquisition in mobile crowdsensing over fog computing platforms for smart cities," *IEEE Transactions on Sustainable Computing*, vol. 2, no. 4, pp. 345–358, 2017.

[32] P. A. Flach, "The geometry of roc space: understanding machine learning metrics through roc isometrics," in *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*, 2003, pp. 194–201.

[33] D. Freedman and P. Diaconis, "On the histogram as a density estimator: L 2 theory," *Zeitschrift für Wahrscheinlichkeitstheorie und verwandte Gebiete*, vol. 57, no. 4, pp. 453–476, 1981.

[34] J. Furch, T. Turo, Z. Krobot, and J. Stastny, "Using telemetry for maintenance of special military vehicles," in *International Conference on Modelling and Simulation for Autonomous Systems*. Springer, 2017, pp. 392–401.

[35] ——, "Using telemetry for maintenance of special military vehicles," in *Modelling and Simulation for Autonomous Systems*, J. Mazal, Ed. Cham: Springer International Publishing, 2018, pp. 392–401.

[36] J. Gama, *Knowledge discovery from data streams*. Chapman and Hall/CRC, 2010.

[37] V. Garcia-Font, C. Garrigues, and H. Rifà-Pous, "A comparative study of anomaly detection techniques for smart city wireless sensor networks," *Sensors*, vol. 16, no. 6, p. 868, 2016.

[38] H. Gomaa, *Real-time software design for embedded systems*. Cambridge University Press, 2016.

[39] C. Goutte and E. Gaussier, "A probabilistic interpretation of precision, recall and f-score, with implication for evaluation," in *European Conference on Information Retrieval*. Springer, 2005, pp. 345–359.

[40] B. K. Gowru and P. Potnuri, "Parallel two phase k-means based on mapreduce," *International Journal of Advance Research in Computer Science and Management Studies*, vol. 22, pp. 3–11, 2015.

[41] I. Guyon and A. Elisseeff, "An introduction to variable and feature selection," *Journal of machine learning research*, vol. 3, no. Mar, pp. 1157–1182, 2003.

[42] H. Hu, Y. Wen, T.-S. Chua, and X. Li, "Toward scalable systems for big data analytics: A technology tutorial," *IEEE access*, vol. 2, pp. 652–687, 2014.

[43] R. J. Hyndman, "The problem with sturges rule for constructing histograms," 1995.

[44] D. Inkpen, J. Ghosh, and D. Lee, "Performance evaluation of information retrieval systems," in *Faculty of Engineering, Power Point Slides*. University of Ottawa, Ottawa, Ontario, Canada.

[45] M. Johanson, S. Belenki, J. Jalminger, M. Fant, and M. Gjertz, "Big automotive data: Leveraging large volumes of data for knowledge-driven product development," *2014 IEEE International Conference on Big Data (Big Data)*, 2014.

[46] M. Johanson and L. Karlsson, "Improving vehicle diagnostics through wireless data collection and statistical analysis," *2007 IEEE 66th Vehicular Technology Conference*, 2007.

[47] V. Karagiannis, P. Chatzimisios, F. Vazquez-Gallego, and J. Alonso-Zarate, "A survey on application layer protocols for the internet of things," *Transaction on IoT and Cloud Computing*, vol. 3, no. 1, pp. 11–17, 2015.

[48] H. Kargupta, V. Puttagunta, M. Klein, and K. Sarkar, "On-board vehicle data stream monitoring using minefleet and fast resource constrained monitoring of correlation matrices," *New Generation Computing*, vol. 25, no. 1, pp. 5–32, 2006.

[49] H. Kargupta, K. Sarkar, and M. Gilligan, "Minefleet®: an overview of a widely adopted distributed vehicle performance data mining system," in *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2010, pp. 37–46.

[50] J. D. Kelleher, B. Mac Namee, and A. DArcy, "Fundamentals of machine learning for predictive analytics," 2015.

[51] P. Killeen, B. Ding, I. Kiringa, and T. Yeap, "Iot-based predictive maintenance for fleet management," *Procedia Computer Science*, vol. 151, pp. 607–613, 2019.

[52] P. Killeen and A. Parvizimosaed, "An ahp-based evaluation of real-time stream processing technologies in iot," in *Course report, Faculty of Engineering, CSI5311*. University of Ottawa, Ottawa, Ontario, Canada, Winter 2018, 2018. [Online]. Available: https://www.mudlakebiodiversity.ca/papers/ahp-based-evaluation-iot-2018.pdf

[53] J. Kim, H. Hwangbo, and S. Kim, "An empirical study on real-time data analytics for connected cars: Sensor-based applications for smart cars," *International Journal of Distributed Sensor Networks*, vol. 14, no. 1, p. 1550147718755290, 2018.

[54] S. Kumar, E. Dolev, and M. Pecht, "Parameter selection for health monitoring of electronic products," *Microelectronics Reliability*, vol. 50, no. 2, p. 161168, 2010.

[55] D. Kwon, M. R. Hodkiewicz, J. Fan, T. Shibutani, and M. G. Pecht, "Iot-based prognostics and systems health management for industrial applications," *IEEE Access*, vol. 4, pp. 3659–3670, 2016.

[56] Y. Lei, N. Li, S. Gontarz, J. Lin, S. Radkowski, and J. Dybala, "A model-based method for remaining useful life prediction of machinery," *IEEE Transactions on Reliability*, vol. 65, no. 3, p. 13141326, 2016.

[57] H. Liao and J. Lee, "Predictive monitoring and failure prevention of vehicle electronic components and sensor systems," *SAE Technical Paper Series*, Mar 2006.

[58] Libelium. (2018, Oct.) Lorawan coverage for latam and asia-pacific on its iot sensor platform. [Online]. Available: http://www.libelium.com/libelium-expands-lorawan-coverage-for-latam-and-asia-pacific-on-its-iot-sensor-platform/?utm_source=NewsletterLB&utm_medium=Email&utm_campaign=NLB-301018

[59] L. Liu, Y. Peng, and D. Liu, "Feser: A data-driven framework to enhance sensor reliability for the system condition monitoring," *Microelectronics Reliability*, vol. 64, p. 681687, 2016.

[60] L. Lui, S. Wang, D. Liu, Y. Zhang, and Y. Peng, "Entropy-based sensor selection for condition monitoring and prognostics of aircraft engine," *Microelectronics Reliability*, vol. 55, p. 20922096, 2015.

[61] H. Madsen, B. Burtschy, G. Albeanu, and F. Popentiu-Vladicescu, "Reliability in the utility computing era: Towards reliable fog computing," in *Systems, Signals and Image Processing (IWSSIP), 2013 20th International Conference on*. IEEE, 2013, pp. 43–46.

[62] K. Martin and M. Marzi, "Diagnostics of a coolant system via neural networks," *Proceedings of the Institution of Mechanical Engineers, Part I: Journal of Systems and Control Engineering*, vol. 213, no. 3, pp. 229–242, 1999.

[63] R. A. Maxion and R. R. Roberts, *Proper use of ROC curves in Intrusion/Anomaly Detection*. University of Newcastle upon Tyne, Computing Science, 2004.

[64] M. Mesgarpour, D. Landa-Silva, and I. Dickinson, "Overview of telematics-based prognostics and health management systems for commercial vehicles," in *International Conference on Transport Systems Telematics*. Springer, 2013, pp. 123–130.

[65] Meshcentral. Meshcentral. [Online]. Available: https://www.meshcentral.com

[66] ModMyPi. (2018) Ups pico installation. [Online]. Available: https://github.com/modmypi/PiModules/wiki/UPS-PIco-Installation

[67] D. C. Montgomery and G. C. Runger, *Applied statistics and probability for engineers*. John Wiley & Sons, 2010.

[68] A. Mosallam, K. Medjaher, and N. Zerhouni, "Data-driven prognostic method based on bayesian approaches for direct remaining useful life prediction," *Journal of Intelligent Manufacturing*, vol. 27, no. 5, pp. 1037–1048, 2016.

[69] N. Najjar, S. Gupta, J. Hare, S. Kandil, and R. Walthall, "Optimal sensor selection and fusion for heat exchanger fouling diagnosis in aerospace systems," *IEEE Sensors Journal*, vol. 16, no. 12, pp. 4866–4881, 2016.

[70] A. H. Ngu, M. Gutierrez, V. Metsis, S. Nepal, and Q. Z. Sheng, "Iot middleware: A survey on issues and enabling technologies," *IEEE Internet of Things Journal*, vol. 4, no. 1, pp. 1–20, 2017.

[71] H. Norrman, "Estimating p-values for outlier detection," http://urn.kb.se/resolve?urn=urn:nbn:se:hh:diva-25662, 2014.

[72] M. S. Obaidat and P. Nicopolitidis, *Smart cities and homes: Key enabling technologies*. Morgan Kaufmann, 2016.

[73] B. Omoniwa, R. Hussain, M. A. Javed, S. H. Bouk, and S. A. Malik, "Fog/edge computing-based iot (feciot): Architecture, applications, and research issues," *IEEE Internet of Things Journal*, 2018.

[74] B. S. Parker, *Sluicebox: Semi-supervised learning for label prediction with concept evolution and tracking in non-stationary data streams*. The University of Texas at Dallas, 2014.

[75] Y. Peng, M. Dong, and M. J. Zuo, "Current status of machine prognostics in condition-based maintenance: a review," *The International Journal of Advanced Manufacturing Technology*, vol. 50, no. 1-4, pp. 297–313, 2010.

[76] R. Pi. Raspberry pi gpio documentation. [Online]. Available: https://www.raspberrypi.org/documentation/usage/gpio/

[77] PiModules and ModMyPi. (2015) Uninterruptible power supply with peripherals and i2c control interface. [Online]. Available: https://pimodules.com/_pdf/UPS_PIco_Manual.pdf

[78] ——. (2017) Ups pico hv3.0 hat. [Online]. Available: https://pimodules.com/_pdf/13_0x40_W_UPS_PIco_HV3.0.pdf

[79] R. Prytz, S. Nowaczyk, T. Rgnvaldsson, and S. Byttner, "Predicting the need for vehicle compressor repairs using maintenance records and logged vehicle data," *Engineering Applications of Artificial Intelligence*, vol. 41, p. 139150, 2015.

[80] S. Ramanujam, M. Gautam, and E. Vasiete, "Predictive maintenance for connected cars," May 2016. [Online]. Available: https://content.pivotal.io/blog/the-data-science-behind-predictive-maintenance-for-connected-cars

[81] T. Rögnvaldsson, H. Norrman, S. Byttner, and E. Järpe, "Estimating p-values for deviation detection," in *Self-Adaptive and Self-Organizing Systems (SASO), 2014 IEEE Eighth International Conference on*. IEEE, 2014, pp. 100–109.

[82] T. Rögnvaldsson, S. Nowaczyk, S. Byttner, R. Prytz, and M. Svensson, "Self-monitoring for maintenance of vehicle fleets," *Data Mining and Knowledge Discovery*, vol. 32, no. 2, pp. 344–384, Mar 2018. [Online]. Available: https://doi.org/10.1007/s10618-017-0538-6

[83] H. Said, T. Nicoletti, and P. Perez-Hernandez, "Utilizing telematics data to support effective equipment fleet-management decisions: Utilization rate and hazard functions," *Journal of Computing in Civil Engineering*, vol. 30, no. 1, p. 04014122, 2016.

[84] U. Shafi, A. Safi, A. R. Shahid, S. Ziauddin, and M. Q. Saleem, "Vehicle remote health monitoring and prognostic maintenance system," *Journal of Advanced Transportation*, vol. 2018, p. 110, 2018.

[85] SIMCom. (2013) Sim800 series at command manual. [Online]. Available: https://cdn-shop.adafruit.com/datasheets/sim800_series_at_command_manual_v1.01.pdf

[86] M. Stonebraker, U. Çetintemel, and S. Zdonik, "The 8 requirements of real-time stream processing," *ACM Sigmod Record*, vol. 34, no. 4, pp. 42–47, 2005.

[87] M. Svensson, S. Byttner, and T. Rognvaldsson, "Self-organizing maps for automatic fault detection in a vehicle cooling system," *2008 4th International IEEE Conference Intelligent Systems*, 2008.

[88] M. Svensson, S. Byttner, and T. Rgnvaldsson, "Vehicle diagnostics method by anomaly detection and fault identification software," *SAE International Journal of Passenger Cars - Electronic and Electrical Systems*, vol. 2, no. 1, p. 352358, 2009.

[89] A. Tahat, A. Said, F. Jaouni, and W. Qadamani, "Android-based universal vehicle diagnostic and tracking system," *2012 IEEE 16th International Symposium on Consumer Electronics*, 2012.

[90] D. Thangavel, X. Ma, A. Valera, H.-X. Tan, and C. K.-Y. Tan, "Performance evaluation of mqtt and coap via a common middleware," in *Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), 2014 IEEE Ninth International Conference on*. IEEE, 2014, pp. 1–6.

[91] A. Theissler, "Detecting known and unknown faults in automotive systems using ensemble-based anomaly detection," *Knowledge-Based Systems*, vol. 123, pp. 163 – 173, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950705117301077

[92] L. M. Vaquero and L. Rodero-Merino, "Finding your way in the fog: Towards a comprehensive definition of fog computing," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 5, pp. 27–32, 2014.

[93] P. Verma and S. K. Sood, "Fog assisted-iot enabled patient health monitoring in smart homes," *IEEE Internet of Things Journal*, 2018.

[94] W. Voss, *A comprehensible guide to controller area network*. Copperhill Media, 2010.

[95] ——, *A comprehensible guide to J1939*. Copperhill Media, 2010.

[96] ——. (2016) Sae j1939 ecu simulator and data monitor for raspberry pi. [Online]. Available: https://copperhilltech.com/blog/sae-j1939-ecu-simulator-and-data-monitor-for-raspberry-pi/

[97] R. P. Walter and E. P. Walter, *Data Acquisition from HD Vehicles Using J1939 CAN Bus*. Society of Automotive Engineers, 2016.

[98] T. Wang, J. Yu, D. Siegel, and J. Lee, "A similarity-based prognostics approach for remaining useful life estimation of engineered systems," in *Prognostics and Health Management, 2008. PHM 2008. International Conference on*. IEEE, 2008, pp. 1–6.

[99] Wikipedia. Freedman diaconis rule. [Online]. Available: https://en.wikipedia.org/wiki/Freedman%E2%80%93Diaconis_rule

[100] ——. Power set. [Online]. Available: https://en.wikipedia.org/wiki/Power_set

[101] W. Wu and L. Gruenwald, "Research issues in mining multiple data streams," *Proceedings of the First International Workshop on Novel Data Stream Pattern Mining Techniques - StreamKDD 10*, 2010.

[102] J. Xu, Y. Wang, and L. Xu, "Phm-oriented integrated fusion prognostics for aircraft engines based on sensor data," *IEEE Sensors Journal*, vol. 14, no. 4, pp. 1124–1132, 2014.

[103] R. C. M. Yam, P. Tse, L. Li, and P. Tu, "Intelligent predictive decision support system for condition-based maintenance," *The International Journal of Advanced Manufacturing Technology*, vol. 17, no. 5, p. 383391, Jan 2001.

[104] Y. Zhang, G. W. Gantt, M. J. Rychlinski, R. M. Edwards, J. J. Correia, and C. E. Wolf, "Connected vehicle diagnostics and prognostics, concept, and initial practice," *IEEE Transactions on Reliability*, vol. 58, no. 2, pp. 286–294, 2009.

[105] A. Zimek, E. Schubert, and H.-P. Kriegel, "A survey on unsupervised outlier detection in high-dimensional numerical data," *Statistical Analysis and Data Mining*, vol. 5, no. 5, p. 363387, 2012.

[106] P. Zschech, "A taxonomy of recurring data analysis problems in maintenance analytics," 2018.