UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

Integration of multiple data sources and dashboard for a remote monitoring system

Inês Silva Sousa

Mestrado em Informática

Dissertação orientada por:
Prof. Doutor José Manuel da Silva Cecílio
Prof. Doutor António Casimiro Ferreira da Costa

2022

Agradecimentos

Gostaria primeiramente de agradecer à minha família, nomeadamente aos meus pais que nos momentos mais difíceis durante o mestrado e no desenvolvimento desta tese sempre me apoiaram e me fizeram continuar mesmo quando queria desistir.

Ao meu namorado Luís que sempre ouviu as minhas reclamações e soube sempre o que dizer para que continuasse a acreditar que conseguiria chegar ao fim do trabalho.

Aos meus amigos, David Silva, Eduardo Pereira, João Ye e Raúl Koch que estiveram sempre ao meu lado durante os meus 5 anos académicos e fizeram-me sentir que não estava sozinha nos momentos mais difíceis. Obrigada pela vossa companhia.

Aos meus orientadores Professor António Casimiro e Professor José Cecílio pelo acompanhamento ao longo do projeto.

Gostaria de agradecer às pessoas envolvidas no projeto do AQUAMON do Laboratório Nacional de Engenharia Civil, nomeadamente à Marta Rodrigues e ao Gonçalo Jesus pelo apoio que me deram ao longo deste ano.

Agradecer também ao Laboratório de Sistemas Informáticos de Grande Escala (LASIGE) pela oportunidade de desenvolver a minha tese de mestrado.

Por fim, agradecer a todos os restantes que fizeram parte do meu percurso académico.

Este trabalho foi suportado pela Fundação para a Ciência e a Tecnologia (FCT) através do projeto AQUAMON (ref. PTDC/CCI-COM/30142/2017) e da unidade de investigação LASIGE (ref. UIDB/00408/2020 e ref. UIDP/00408/2020), e pelo programa de inovação e investigação Horizon 2020 da União Europeia através do projeto VEDLIoT (grant agreement No 957197).

Para o meu avô

Abstract

Wireless sensor networks used in aquatic environments for continuous monitoring are typically subject to physical or environmental factors that create anomalies in collected data. A possible approach to identify and correct these anomalies is to use artificial neural networks, as the previously proposed ANNODE (Artificial Neural Network-based Outlier Detection), to improve the data quality.

The ANNODE framework uses neural networks trained to detect outliers in a time series dataset. The framework is split into two parts, one for off-line training of neural networks, and the other for real-time processing of incoming sensor data and outlier detection.

This work proposes ANNODE+, which extends the ANNODE framework by detecting missing data in addition to outliers, and by handling multiple simultaneous time series inputs, as long as the target is to detect anomalies in those time series. A new design and implementation were planned for ANNODE+, as well as an interface in which the framework's behaviour can be seen in real-time, enabling the user to observe the detected anomalies and their correction.

ANNODE+ also correlates different variables that might contribute to changes in water quality, and can identify real events that look like anomalies, thus avoiding false positives. In addition, a REST API was also implemented to fetch all stored measurements collected by sensors.

This work was developed in the scope of the AQUAMON project, whose objective is to develop a dependable platform based on wireless sensor networks to monitor aquatic environments. Two datasets are used to evaluate the ANNODE+ capabilities. One dataset was built from a sensor deployment in Seixal's bay, Portugal. This dataset includes measurements of water level, temperature, and salinity. The second dataset was collected from the SATURN Observation Network, which comprises water temperature measurements.

In terms of results, the new ANNODE+ framework performs as intended, detecting anomalies and correcting them successfully. Every received and corrected measurement is stored in a database which is then accessed to graphically show the framework's behaviour through a specifically built interface.

Keywords: Neural Networks, Environmental Monitoring, Sensor Networks, Forecasting, Data Quality

Resumo Alargado

Manter um bom nível de qualidade da água é essencial para a preservação da fauna e flora aquática, bem como de todos os animais e organismos que dela depende. Com a atual escassez de água e com os maiores níveis de poluição da mesma, é crucial monitorizar os níveis de qualidade da mesma. A Internet das Coisas (IoC) e as redes de sensores sem fios (RSSF) surgem como algumas das ferramentas fundamentais e mais utilizadas em sistemas que monitorizam a qualidade da água. As RSSF são redes com sensores dedicados que detetam fenómenos ou eventos específicos e têm sido utilizadas para monitorizar remotamente vários ambientes aquáticos como rios, costas, lagos e baías. Possuem a vantagem de fornecer dados em tempo-real, o que é vital para alertar os sistemas de emergência ou as autoridades em caso de situações atípicas.

Uma vez que as RSSF estão constantemente a ser afetadas por fatores físicos ou ambientais que podem criar incoerências nos dados recolhidos a fim de fornecer a fiabilidade necessária, essas soluções necessitam de ser apoiadas por plataformas que detetam dados defeituosos ou até a omissão dos mesmos. Os dados recolhidos pelos sensores podem sofrer irregularidades devido a mudanças súbitas no estado da água, graças a um defeito no *hardware* do sensor, grandes rajadas de vento, objetos que obstruem a ligação entre o sensor e a *gateway*, para onde envia os seus dados, entre outros fatores físicos.

Com intuito de ajudar os gestores a analisar os dados em tempo real, a utilização de interfaces intuitivos que exibam facilmente todas as falhas que são detetadas e as suas respetivas correções são necessárias. Gráficos e diagramas permitem apresentar diferentes tipos de dados sem deixar o utilizador perdido com grandes porções de dados.

No entanto, apesar de ser um problema bastante atual e cada vez mais importante, é escasso o número de soluções existentes para resolver a falta de monitorização de dados em ambientes aquáticos. SmartCoast foi um projeto inovador nos anos 2000 que recolhia dados de um sensor alocado em Cork, Irlanda. Consistia numa RSSF que monitorizavam a qualidade da água naquele local. Contudo, este projeto não tinha como garantir que os dados que recolhia eram válidos e por isso necessitava de um processo que os avaliasse constantemente.

Os projetos mais relevantes em Portugal que se seguiram fazem parte de um esforço coletivo do Laboratório Nacional de Engenharia Civil (LNEC) que elaborou várias iterações com o mesmo cerne até chegar a uma plataforma de desenvolvimento de software, a *framework* ANNODE. O objetivo final seria a implementação de um sistema capaz de receber e processar dados em tempo real garantindo a qualidade dos dados e consequentemente, a garantia da fiabilidade dos dados da qualidade da água onde o sistema estaria ativo.

Por vezes os ambientes aquáticos possuem condições adversas que podem influenciar a recolha de dados de forma coerente. Desta forma, é necessário garantir a qualidade dos dados que recebemos e

que processamos. Este projeto dá continuidade ao trabalho realizado pelo colega J. Penim [37] onde a *framework* ANNODE foi estendida e melhorada de modo a proceder à recolha de dados em tempo real. A *framework* foi desenvolvida de modo a receber dados e detetar *outliers*, *drifts* e *offsets* com a ajuda de redes neuronais. Esta faz previsões com base em valores anteriormente recolhidos. De seguida, o valor recebido passa por um bloco de qualidade onde o erro quadrático é calculado para verificar se existe ou não uma alta probabilidade do valor recebido ser incorreto. Em caso afirmativo, o valor recebido é substituído pelo valor previsto. Desta forma, a *framework* deteta *outliers*, *drifts* e *offsets* corrigindo-os com valor de acordo com o esperado.

A *framework* ANNODE oferece capacidades para processar dados de um único sensor (fonte de dados) ou de várias fontes de dados. No entanto, a *framework* mostra várias limitações quando uma única fonte de dados é utilizada, uma vez que não existem possibilidades de estabelecer correlações de dados entre sensores. Esta *framework* foi também implementada para um conjunto de dados específico, os dados do *Saturn Observatory Network*, o que impossibilita o uso de outros conjuntos de dados.

No contexto desta tese, e tendo em conta a necessidade de plataformas deste tipo para lidar com dados provenientes de RSSF, uma nova versão da *framework* foi desenvolvida com novos mecanismos de maneira a ser mais útil nos cenários de monitorização remotos. Como complemento à *framework* existente, uma estrutura será implementada com o objetivo de dar apoio à *framework* e a outros serviços importantes a desenvolver como uma interface gráfica.

Assim, os objetivos delineados para esta tese consistem no melhoramento funcional da *framework*, uma nova organização estrutural para melhor compreensão da mesma, uma implementação de uma API e o desenvolvimento de raiz de um *dashboard* de maneira a integrar todos os componentes da plataforma numa interface gráfica e acessível ao utilizador.

Para o melhoramento do funcionamento da *framework* foi necessário entender a versão anterior da mesma de modo a criar novas funcionalidades e modificar outras já implementadas de maneira a otimizar o código. A fim de melhorar a organização estrutural do código usado durante a implementação da versão anterior de ANNODE, foi desenhada uma arquitetura que possibilita a adição de novas funcionalidades de forma simples e rápida.

Com o objetivo de ter uma plataforma sempre em funcionamento e que possa estar à escuta de novas conexões de maneira a conseguir processar dados, é necessário ajustar a *framework* para um funcionamento contínuo e que seja confiável num ambiente *online*.

A existência de vários componentes díspares onde a comunicação entre os mesmos era importante, implementou-se um ambiente de virtualização através de microserviços e imagens da ferramenta Docker, necessário à utilização simultânea de diferentes serviços virtuais. Deste modo, existem serviços para cada elemento essencial nesta plataforma, nomeadamente, serviços para a *framework* em si, para a interface gráfica, para a API e outras tecnologias como o Grafana e a base de dados MySQL que são utilizadas e que necessitam de acesso constante.

Este projeto está dividido em duas fases. A primeira fase consiste em compreender e alargar a *framework* ANNODE. É essencial compreender como funciona a *framework*, como deteta *outliers*, como é que uma previsão é calculada e como são substituídos os valores considerados como anomalias. É nesta fase que exploramos novas abordagens a adicionar à *framework* de modo a complementar as funcionalidades existentes.

De seguida, é necessário englobar o envio de dados recolhidos por sensores do LNEC para a *framework* e observar o seu comportamento, generalizar a *framework*, definir de fatores que fazem uma medição um *outlier*, reformular a *framework* para um uso mais compreensível e um melhor funcionamento, implementar novos métodos para a deteção de anomalias, mais especificamente, omissão de dados ou dados em falta.

A segunda fase consiste na implementação de um *dashboard* para o sistema de monitorização remota, onde uma arquitetura de sistema necessita de ser construída de modo a apoiar a interface gráfica. Esta deve mostrar ao utilizador, em tempo real, dados e as suas anomalias, caso sejam detetadas.

No final, esta solução mostrou ser fiável no contexto desta tese. Obtivemos resultados bastante promissores que melhoram esta contribuição na inovação de soluções para este tipo de problema, sendo possível a deteção de anomalias como *outliers* e omissões de dados em tempo real, assim como a sua correção. Este processamento de dados é valido não só para apenas um sensor, mas também para vários sensores que enviam dados em simultâneo.

Palavras-chave: Redes Neuronais, Monitorização Ambiental, Redes de Sensores, Previsão de Valores, Qualidade de Dados

Contents

List of Figures xvi

List of Tables xix

Acronyms xxi

1 Introduction 1

1.1 Motivation . . . 2
1.2 Objectives and Contributions . . . 3
1.3 Methodology . . . 3
1.4 Document Structure . . . 4

2 Related Work 7

2.1 Data Quality . . . 7
2.2 Sensor Data Quality . . . 8
2.2.1 Types of errors in sensor data . . . 9
2.2.2 Methods to detect anomalies . . . 10
2.2.3 Methods to detect and correct anomalies . . . 11
Principal Component Analisys (PCA) . . . 11
Artificial Neural Network (ANN) . . . 11
Bayesian Network . . . 12
2.3 Dependable Monitoring Systems . . . 12
2.4 Neural Networks . . . 13
2.4.1 Feedforward neural networks . . . 13
Single-layer Perceptron . . . 14
Multi-layer Perceptron . . . 14
Radial basis function artificial neural networks . . . 15
2.4.2 Recurrent neural networks . . . 15
2.4.3 Convolution Neural Network . . . 16
2.4.4 Comparison between CNNs, RNNs and MLPs . . . 16
2.5 Technologies . . . 17
2.5.1 Type of neural network . . . 17
2.5.2 Implementation technologies . . . 18
Docker . . . 18

Grafana . . . 19
2.6 AQUAMON . . . 19
2.7 Summary . . . 20

3 Proposal **23**
3.1 System Architecture . . . 23
3.2 ANNODE+ Architecture . . . 24
3.2.1 Training Block . . . 25
Data processing configuration file . . . 26
Training configuration file . . . 27
Entry vector's creation . . . 29
Align times . . . 30
Training neural networks . . . 31
3.2.2 Execution Block . . . 32
Execution configuration file . . . 33
Communication with the framework . . . 34
Communication service . . . 34
Processing service . . . 34
Omission detection . . . 35
Prediction block . . . 35
Quality assurance block and outlier detection . . . 36
3.2.3 Graphical Interface . . . 38
3.3 API . . . 39
3.3.1 Documentation . . . 39
3.4 Summary . . . 39

4 Implementation **41**
4.1 Data processing . . . 41
4.1.1 Input data structure . . . 41
4.2 Neural network training . . . 42
4.3 Framework overview . . . 44
4.3.1 Server . . . 44
SensorHandler . . . 45
SensorData . . . 45
PredictionBlock . . . 46
QualityBlock . . . 46
4.4 Docker . . . 46
4.4.1 Docker flow . . . 46
4.4.2 Containers and Images . . . 47
4.5 Dashboard . . . 49
4.5.1 Sensor configuration . . . 50
4.5.2 MySQL Database . . . 50

4.5.3 Grafana 51
4.6 API Implementation 52
4.7 Framework's and Docker execution 53
4.7.1 Framework directory 54
4.8 Summary 55

5 Results 57
5.1 Data 57
5.2 Framework validation with previous work 58
5.2.1 Outlier detection 58
5.2.2 Omission failure detection 59
5.2.3 Comparisons between versions 60
5.3 Validation with LNEC's data 61
5.3.1 Outlier and omission detection 61
5.3.2 Different approaches for entry vector creation 62
5.4 Dashboard data and real time validation 63
5.5 API results 63
5.6 Summary 65

6 Conclusion 67
6.1 Future work 68
6.2 Publications 68

References 74

List of Figures

2.1 Example of a linear classifier [16] 14
2.2 MLP Network 15

3.1 System architecture 24
3.2 Framework's overview 25
3.3 Training block architecture 26
3.4 Training sets configuration file 27
3.5 Training configuration file 28
3.6 Example of the entry vector's construction using the target sensor in an exponential approach [37] 29
3.7 Linear approach (left) and last ten approach (right) 30
3.8 Alignment of Times [37] 31
3.9 Cumulative Density Function of the distribution probability of the square errors 31
3.10 Execution block architecture 33
3.11 Execution configuration file 33
3.12 Omission detection 35
3.13 Entry vector creation with multiple sensors [37] 36
3.14 Outlier detection 37
3.15 Prototype of the configuration page 38
3.16 Prototype of the main page 38

4.1 Example of *.csv* file structure 42
4.2 Example of *.json* file structure 42
4.3 Initial input for training depending on model 43
4.4 CDF of the probability distribution of the square errors 43
4.5 Log-logistic probability distribution of the square errors 43
4.6 Sequence diagram of *app_server.py* 45
4.7 Docker flow 47
4.8 Docker-compose file example 48
4.9 Grafana Dockerfile 49
4.10 Dashboard main page 50
4.11 Database Relation schema 51
4.12 Grafana directory 52
4.13 File structure 53

5.1 SATURN Observation Network . . . 57
5.2 *Desdemona* raw measurements . . . 59
5.3 *Tansy Point* raw measurements . . . 59
5.4 *Lower Sd* raw measurements . . . 59
5.5 *Jetty A* raw measurements . . . 59
5.6 Representation of missing data periods . . . 60
5.7 Framework's outcome with data from the Seixal bay . . . 61
5.8 Forecast calculation with all approaches . . . 63
5.9 All Measurements panel in Grafana . . . 63

List of Tables

2.1 Main methods of fault detection used in single-sensor and multi-sensor situations [28] . . 9
2.2 Most Common Types of Error in Sensor Data [44] 9
2.3 Most Common Methods for Error Detection in Sensor Data [44] 10
2.4 MLPs vs RNNs vs CNNs 17

5.1 ANNODE+ results 59
5.2 Comparison for *Jetty A* and *Lower Sd* sensors 60
5.3 Comparison for *Desdemona* and *Tansy* sensors 60
5.4 Results from LNEC's sensor with two types of variables 62

Acronyms

ANNODE Artificial Neural Network-based Outlier Detection

REST Representational State Transfer

API Application Programming Interface

IdC Internet das Coisas

IoT Internet of Things

RSSF Redes de Sensores Sem Fios

LNEC Laboratório Nacional de Engenharia Civil

WSNs Wireless Sensor Networks

ANNs Artificial Neural Networks

PCA Principal Component Analysis

AANN Auto-Associative Neural Network

ML Machine Learning

AI Artificial Intelligence

MLP Multi-Layer Perceptron

RBF Radial Basis Function

RNNs Recurrent Neural Networks

CNNs Convolution Neural Networks

NaN Not a Number

CDF Cumulative Distribution Function

TCP Transmission Control Protocol

DR Detection Ratio

FPR False-Positives Ratio

Chapter 1

Introduction

Maintaining good water quality is vital for the aquatic fauna and flora and our life quality. It has become a scarce resource, so it is crucial to monitor it. Internet of Things (IoT) and Wireless Sensor Networks (WSNs) are essential in monitoring and inspecting their quality. WSNs are networks with dedicated sensors that detect specific phenomena or events. WSNs have been used to monitor many aquatic environments such as rivers remotely, coasts, lakes and bays [30, 36]. WSNs can provide real time data, which is vital to alert emergency systems or authorities in atypical situations.

Given that WSNs and their sensors are exposed to physical or environmental factors that often create anomalies in collected data, existing solutions can benefit from platforms for detecting erroneous data or data omissions to provide the required reliability.

In this work, we propose the ANNODE+ framework, an artificial neural network-based framework for online data quality assurance. Taking inspiration from ANNODE, an outlier detection framework based on Artificial Neural Networks (ANNs) previously proposed by Jesus et al [30]. The new version of the framework is more generic, extending previous work with additions of new failure detection methods and a new implementation. With the support of ANNs, the framework considers incoming measurements as time series (e.g., temperature values over time), predicting future values in the series. Each received measurement goes through a set of blocks to determine if it is an outlier, to estimate its quality, and, if considered an outlier, to replace it with a corrected measurement.

Our framework was designed for the online processing of incoming sensor measurements, and implemented with real time concerns in mind, to reduce the time taken to process each incoming measurement and avoid arbitrarily large processing times.

It offers capabilities to deal with a single sensor (data source) or multiple sensors providing correlated measurements. The ability to detect outliers can be significantly improved when correlated data sources are available. Measurements can be correlated if different variables impact one another, e.g., salinity levels can change temperature levels. If more than one data source is available, some events can be explained as incidents. For instance, if it is detected a change in water levels, this change will also be detected in other sensors. However, if an event is detected by only one data source, it is most likely an anomaly.

1.1 Motivation

As referenced before, the remote management of water quality in strategic places can improve wildlife as well as life quality for the ones that live nearby. It can help identify the water's state and notify authorities in case of unusual events, thus preventing negative environmental impacts. Furthermore, we know that aquatic environments suffer harsh conditions that can sometimes clout collected data from sensors. Wireless Sensor Networks (WSNs) have been used in these types of environments, one solution being the ANNODE framework by [37] in which this thesis extends.

When authorities are involved in these projects, data quality must be ensured to avoid false alarms. Because sensors are located in a physical environment, it is prone to suffer from physical events, such as wind gusts and loose debris that can damage the equipment and/or alter collected data. However, few reliable, dependable systems can deal with this problem accurately. With this lack of systems, collected data becomes even more faulty, which may lead to various hazards in the long term derived from bad decision-making from authorities.

To detect if a measurement is an anomaly or not, it is important to understand the context in which it happened. Some random events could be happening that will have an impact on the sensors. When this happens, we need to analyse if all sensors in the WSN detected that event. To do this, we rely on artificial intelligence to catch these events that can originate erroneous data. With the newest techniques available in neural networks, it is possible to create a platform that can manage complex data structures without going overboard with resources.

Previous work shows improvement of outlier detection in an online environment (real time) while calculating forecasts with neural networks for the Saturn Observation Network [9] dataset. Multi-layer perceptron neural networks were used for the generation of accurate forecasts. In order to calculate them, prediction models were trained based on previously fetched data that should be correct covering a whole year. This helped to calculate predictions in any season of the year making the forecasts as accurate as possible, consequently detecting real outliers thus reducing the false-positive outliers ratio [21] had.

This project is being developed alongside the Laboratório Nacional de Engenharia Civil (LNEC) [40], which provides us with water datasets of the Seixal bay in Portugal. These datasets consist of variables such as water temperature, salinity levels, pH, chlorophyll, turbidity, dissolved oxygen and oxygen saturation. Because the previous version only supported data from a specific dataset, this work will accept more datasets by generalizing the framework to detect anomalies in water environments with one or multiple sensors. Moreover, we will extend failure detection to forecast measurements in case of missing data. In addition, we will implement a new architecture in the framework improving the workflow for future iterations.

Since the human brain has difficulty analyzing data on a sheet, graphical data representation can go a long way. Dashboards are usually developed to create a system where users can visualize data by graphs with stored data. We will develop a graphical interface to help visualize data from the sensors and the framework's behaviour, thus showing measurements being corrected in real time. To support the system and its features, a structure needs to be built.

1.2 Objectives and Contributions

This thesis aims at integrating multiple data sources and implementing a dashboard for a remote monitoring system. For this, some objectives were outlined:

1. Implement a new architecture for the ANNODE framework;
2. Document the framework;
3. Implement a unification layer of services and APIs;
4. Implement a graphical interface that supports data visualization in the platform.

To do this, the study of RESTful services architecture is needed as well as the understanding of the mechanics of intelligent systems for failure detection.

The main contribution of this project is the extension and implementation of a generic and robust framework that allows real time failure detection in aquatic environments. Another contribution is the implementation of a dashboard for the visualization of data. This project is implemented mainly in Python. To support all parallel running services this platform needs, we will use Docker to create specific containers for each service, such as a database, an API service and web applications.

1.3 Methodology

This project is divided into three phases. The first phase consists of understanding and extending the ANNODE framework. Here we will understand how the framework works, how it detects outliers, how a prediction is calculated and how predictions replace outliers. Then we will explore other approaches to complement the existing ones to make the framework more powerful and valuable for dealing with different data sources.

Secondly, send data collected by LNECs [40] sensors into the framework and observe its behaviour. In order for the framework to work with other datasets, it needs to be generalized. When multiple data sources communicate with the framework, we must define factors that make a measurement an outlier. In other words, we must determine what physical events can lead to erroneous measurements. The framework also needs reformulation for a more understandable code use and proper functioning. New methods for the detection of more anomalies will be implemented, specifically missing data or omissions of data. This second phase involves the implementation of a dashboard for the remote monitoring system, where a system architecture will be built to support the dashboard and its required services for data visualization and data storage. The dashboard should show the user real time data and its anomalies if detected.

The third phase involves developing an API to fetch the framework's results. This API is obviously connected to a database that stores all measurements coming through the framework. There, it stores the raw measurement and its corrected value. The user is then able to get all the information from each measurement that a specific sensor sends through the API.

1.4 Document Structure

The document has the following chapters:

- **Introduction** - Gives an introduction of the problem and methods that can be used to solve the problem and improve the state-of-the-art in the area. Explains previous work done in the AQUAMON project and the status of the ANNODE framework before starting this thesis;
- **Related Work** - Reviews the concepts and methodologies used to deal with sensor data quality, including the types of errors that can occur in these systems. Introduces the state-of-the-art in the area of dependable monitoring systems. Explores the main categories of neural networks. Explains the decision behind the choice of some technologies that were used in this work;
- **Proposal** - Describes the proposed alterations to the framework and explains the new architecture. Explains in detail major parts of the framework and how they work. Introduces the system architecture that supports all the new additions;
- **Implementation** - Describes methods, algorithms, technologies and approaches that are used to tackle the problem. A discussion about their configuration and our decisions is also presented;
- **Results** - Shows results with the current dataset and compares the framework's performance with the previous version;
- **Conclusion** - Discusses the obtained results and explores conclusions that were met throughout the implementation of this work.

Chapter 2

Related Work

There have been many investigations and many different projects using Wireless Sensor Networks. However, there is little information on how sensors detect, and correct anomalies caused by harsh conditions. Section 2.1 will detail some characteristics and dimensions of data quality. Section 2.2 will explain what is sensor data quality and the types we may encounter. It will also explore methods to detect and correct these errors. Section 2.3 describes previous work done by other investigators regarding the topic of dependable monitoring systems. Section 2.4 explores different types of neural networks, their advantages and disadvantages as well as descriptions of different types of architectures. Finally, section 2.5 will explore the different technologies we chose to develop this project and the reason behind that choice. For each topic, we reviewed previous work done by different authors and the current state-of-the-art in the context of this thesis.

2.1 Data Quality

Data quality is a measure that defines if data is suited for its purpose. It follows data quality characteristics such as accuracy, precision, consistency and completeness.

Over the last few years, concern for data quality has increased as a boost of production by the private sector and reliance on secondary data sources have taken place. With this, there is a lot of corrupted data that needs to be managed to decrease or eliminate possible errors. As H. Veregin [46] stated, data can be differentiated in space, time and theme, and so each dimension can identify four characteristics:

- **Accuracy** - The inverse of error, free of error and mistakes;
- **Precision** - The exactness of data;
- **Consistency** - Absence of apparent contradictions;
- **Completeness** - Relationship between the objects in the database and the abstract universe of such objects.

Despite these characteristics, if data is not accessible and interpretable by users, it has little value and, therefore, is poor.

According to [17], there are numerous dimensions of data quality. The main dimensions are:

- **Timeliness** - The user should obtain data within a minimal time delay;

- **Consistency** - Data should be complete and correct;
- **Completeness** - All the components of a datum should be valid;
- **Readability** - Data should be correctly explained.

There is also a third method of evaluating data quality. Richard Y. Wang et al. [47] proposed an evaluation of data by modelling methodologies where the outcome is a quality schema with application requirements and data quality issues that are considered important. This process is described in four steps in [47].

Even with these methods' knowledge, there are some challenges that come with data quality that should not be discarded. These problems usually appear when we are dealing with errors:

- **Complex data structures and different data types increase the difficulty of data integration** - big data sources come from the internet, mobile internet, the Internet of Things, various industries and scientific experimental and observational data such as the SATURN Observation Network [9]. For each of these sources, there are plenty of different types of data which makes data processing difficult for so many categories;
- **Big data volume** - For such big amounts of information, it is difficult to collect, clean and get quality data in a reasonable time;
- **Data changes very fast** - When data is being continuously fetched, it is always changing and so, it is very challenging to process it before the data changes again;
- **Research in big data has only just begun** - There is little information about standards and methods to deal with so much information coming from all directions.

Finally, there are many ways of ensuring data quality, each unique on its own and dependable on the application requirements. The same data can be rich for one application and poor for another. Hence an inquiry should be made to understand which data parameters are important for the project and how to tackle them.

2.2 Sensor Data Quality

Sensor data quality is vital for the project as the framework receives a significant amount of data that can be corrupted due to harsh conditions in aquatic areas. To establish a good database, artificial intelligence is the most common solution to use in these problems. This section is widely influenced by the review and survey from Hui Yie Teh et al. [44], and Gonçalo Jesus et al. [28] respectfully.

First, it is important to differentiate single-sensor validity from multi-sensor fusion validity to understand the requirements in data quality for each of these two cases. Thus, Table 2.1 represents the different methods that allow the detection of possible faults:

Although different approaches and methods can be used depending on how many sensors our system may have, the types of errors and techniques to tackle them are the same regardless of the number of active sensors.

Table 2.1: Main methods of fault detection used in single-sensor and multi-sensor situations [28]

Single-sensor	Multi-sensor
Rule-based methods, determine thresholds or heuristics.	Redundant or correlated data between sensors
Estimation methods considering correlations from sensor data.	Quality-oriented network meta-models
Learning-based methods.	Sensor data fusion methods

2.2.1 Types of errors in sensor data

There are many different types of errors, but they can be generally classified as:

- **Global anomalies** - considered a global anomaly when the data point is fairly different from the dataset at hand;
- **Contextual or conditional anomalies** - these anomalies deviate from other data within the same context;
- **Collective anomalies** - a subset of data is a collective anomaly when it deviates from the rest of the dataset as a whole.

The most frequent errors detected in sensor data are outliers, missing data, bias and drifts. As already discussed herein, the framework detects and corrects most of these common errors. These errors can be caused by changes in the environment, malfunction of the devices or obstructions in the communication between nodes. For instance, during one of the on-field experiments a boat that was passing through blocked the connection between two nodes and the gateway, which can cause missing data for a short period of time. These cases can then lead to incorrect decision-making when creating predictions with the framework.

Hui Yie Teh et al. in [44] searched for keywords in a database with multiple papers and articles. Table 2.2 shows the most frequent types of error found in papers and the number of papers that address the respective error.

Table 2.2: Most Common Types of Error in Sensor Data [44]

Type of error	Total
Outliers	32
Missing Data	16
Bias	12
Drift	12
Noise	8
Constant value	7
Uncertainty	6
Stuck-at-zero	6

Moreover, according to [28] these are the possible errors that can occur in individual sensor measurements:

- **Random errors** - Arbitrary errors;
- **Systematic errors** - These can be of three types: calibration errors, loading errors and environmental errors;
- **Non-systematic reading errors** - These errors happen when a physical event takes place.

The previous errors correspond to errors in measurements obtained by sensors. However, these can always have failures and be broken by significant events like strong winds and floating debris, consequently recording inaccurate data.

2.2.2 Methods to detect anomalies

As previously stated, artificial intelligence and machine learning methods are required to detect anomalies in sensor data. Many techniques, including supervised and unsupervised methods, can detect anomalies based on past data.

Similarly to the types of error, the authors in [44] searched for the most common methods to detect the types of errors shown in Table 2.2. Table 2.3 shows different methods for error detection. The total represents the number of methods based on these error detection methods.

Table 2.3: Most Common Methods for Error Detection in Sensor Data [44]

Method	**Errors addressed**	**Total**
Principal Component Analysis	Outliers, bias, drift, stuck-at-zero	7
Artificial Neural Network	Outliers, bias, drift, constant values, noise, stuck-at-zero, uncertainty	6
Ensemble Classifiers	Outliers, drift, constant values, noise, uncertainty	4
Support Vector Machine	Outliers	2
Clustering	Outliers	2
Ontology / knowledge-based systems	Uncertainty, missing data	2
Univariate autoregressive models	Outliers	1
Statistical Generative Models	Outliers	1
Grey Prediction Model	Outliers, noise, constant values	1
Particle Filtering	Bias, scaling	1
Association Rule Mining	Outliers	1
Bayesian Network	Outliers, noise	1
Euclidean Distance	Outliers	1
Hybrid Methods	Outliers, drift, noise	1

Regardless of all these different methods, there is still space for time-based methods to detect missing data. For instance, if a sensor always sends data every five minutes, new data is expected within that time frame. So if none comes, it can be considered an error.

2.2.3 Methods to detect and correct anomalies

In the same way that there are methods that detect anomalies, the following techniques are more complete considering that they can detect and correct anomalies. These methods are the ones used in the framework of this work. Just like the methods for anomaly detection, these also build models to compare with observed values. Suppose the value is significantly different from the model. In that case, it is considered an anomaly and it is then replaced with a prediction or an estimated value from the model in that instant.

The next list presents three proposed methods by Hui Yie Teh et al. in [44] to detect and correct anomalies in sensor data:

- **Principal Component Analysis (PCA)**;
- **Artifitial Neural Network (ANN)**;
- **Bayesian Network**.

Principal Component Analisys (PCA)

Principal Component Analysis (PCA) is a technique that reduces the dimensionality of datasets by finding patterns in them, such as correlating variables. This technique uses unsupervised learning, which minimizes information loss by converting the observations of correlated variables into new uncorrelated variables with the help of principal orthogonal components analysis.

This method is very useful to find anomalies as it can be used to define multiple variables. According to the authors [44], this method is also useful for edge computing applications because it can perform locally without extra communication overheads. Nonetheless, the training of this method needs to be done with data that is correct, which means, with no anomalies and that is not easy to have.

By using the Squared Prediction Error and the Sensor Validity Index, the PCA method can correct faulty values. To get the most consistent value with the model created, the fault is subtracted by the observed data.

The Squared Prediction Error is a measure that calculates the squared difference between a predicted value and the observed value. This measure is being used in the framework to calculate the error between a prediction and the actual value.

Sensor Validity Index is a measure that calculates the sensor's performance. This measure is able to detect if the sensor is faulty and has low performance or if it is working correctly and according to its requirements.

Artificial Neural Network (ANN)

An Artificial Neural Network (ANN) is a framework that mimics the human brain's neural network. An ANN works almost the same way as a human brain. The main processing power is the neuron. It receives

data and passes it to other neurons that process it until it reaches the output's last neuron. There are three components to an artificial neural network: the input layer, hidden/occult layers and the output layer.

There are many different types of ANNs, but they are all mainly used for pattern recognition. As there are various types, the advantages and disadvantages also vary. Each type has its recommendations for what situations are appropriate to use them. Therefore, when using ANNs, a prior analysis should take place to use the correct type of ANN.

Many authors have introduced different techniques to detect and correct anomalies using artificial neural networks, one of which is the auto-associative neural network (AANN) technique. In this technique, the input and output vectors are identical and use back-propagation or similar learning procedures. Back-propagation is an algorithm that reduces the total loss in each neural network node. The objective is to determine the optimal action corresponding to a particular state by compressing and decompressing data.

Bayesian Network

A Bayesian Network is a probabilistic graphical model made by graphs. It represents knowledge of a particular domain where each node corresponds to a variable and the edge represents the probability for the respective variables. A method was proposed for this model based on a Dynamic Bayesian Network. It learns the normal behaviour of sensors, and if the predicted value diverges from both the model and the actual value it is considered defective.

2.3 Dependable Monitoring Systems

Dependable monitoring systems guarantee the quality of measurements by detecting faults in data. They are reliable and available. In a real time context, dependable systems maintain their reliability within a time period. Usually in these types of systems, when an error occurs the user is notified immediately and the problem is fixed in real time. There have been many experiments with dependable monitoring systems over the years, but few in aquatic environments. In this section, several dependable monitoring systems are reviewed.

In mid-2000's the project SmartCoast [36] was implemented. It consists of a wireless sensor network for water quality monitoring in Cork, Ireland. This project investigates the water's temperature, phosphate, dissolved oxygen, conductivity, pH, turbidity and water level variables, so it is a multi-sensor project. These parameters are almost universal to other projects, being the most required variables for these experiments. The SmartCoast project uses the ZigBee protocol for low power consumption. Because this was the start of projects with water quality monitoring, data quality was lacking.

A few years later, a similar experiment [23] was made in Aveiro, Portugal by LNEC. The authors used and adapted a custom-deployment-based forecasting platform to the Portuguese coast. This allowed us to create numerical models to provide forecasts of sea level variations, currents, temperatures, etc. The authors explain that investigation of applications in harsh environments is minimal. There is a need to explore what happens to sensors and their data in this type of environment. This is important for this thesis as saline waters have significant changes to objects because of their salinity levels. It can also affect data. This experiment was the beginning of developing a real time system for water monitoring.

The platform adapted by LNEC was then implemented in the Tagus estuary [39] with the same sensor architecture and models.

With this platform in the works, there was a need to detect anomalies in the sensors' collected data and ensure its quality. And so, a framework was beginning to be implemented for these cases [29]. This framework should support the detection of erroneous data and the replacement of a value when considered faulty. This framework would later become the framework used and adapted in this thesis.

The authors of [15] created algorithms, using ML methods like the PCA method, to understand how long-term and short-term water demand forecasts can be influenced by different variables such as rain, the hour of the day and air temperature. These algorithms could be proved useful as the authors could easily detect which variables had more impact on the system and hence were able to create a sense of reliability and efficiency for their customers.

Outside the water monitoring systems scope, dependable systems have been developed in many other areas, such as health [20], mobile traffic management [32], construction sites [35] and many others. It is clear, that dependable systems are needed and used in other fields of study. They all use machine learning models and IoT services to accomplish their goals, which is very similar to the dependable systems presented before.

2.4 Neural Networks

Neural networks, also known as Artificial Neural Networks (ANNs), are machine learning algorithms that simulate a human brain by trying to detect specific patterns in data to classify and solve problems. An ANNs comprises at least three types of layers: an input layer, one or more hidden layers and one output layer. Each layer has multiple artificial neurons with associated weights and a threshold. If the output of any neuron surpasses the respective threshold, data is passed to the next layer.

As neural networks are a type of machine learning algorithms, they can have two types of learning:

- **Supervised learning** - This type of learning is a process where neural networks are trained by learning how data is classified when input and output are given. This means that data needs to be labelled. With successive iterations, the model fines its accuracy.

- **Unsupervised learning** - This technique is the opposite of supervised learning, where the model is only exposed to input data. It learns by identifying patterns and behaviours in data.

There are also a lot of different types and architectures of neural networks that are suited for simple and complex problems [25]. These problems can vary from failure detection [34], forecasting [12, 41], image recognition [24] and classification problems [48].

2.4.1 Feedforward neural networks

Feedforward neural network is a type of neural network that moves data forward between the input, hidden and output nodes without forming a cycle [14]. Here, data moves in only one direction (forward) never going backward, making it the simplest form of a neural network. Herein, we will explore some types of feedforward neural networks such as single-layer or multi-layer perceptron and radial basis function networks.

Single-layer Perceptron

A single-layer perceptron has no hidden layer, thus there is only one input layer and one output layer. The output layer is calculated with the sum of the value's respective weight [38]. If this sum surpasses a threshold, the value produced will be 1, otherwise, it is 0. This is defined as:

$$H(x) = \begin{cases} 1 & x \geq 0 \\ 0 & x < 0 \end{cases}$$

Because of this definition, these types of neural networks are used as data classifiers. Since there are only two classes, this type of classification is also known as a linear classifier. Figure 2.1 represents a linear classifier where the squares are evaluated as 0 and the circles as 1 [16]. The line that separates the two classes is called the classification boundary. Depending on how the AI learned to classify values, there can be multiple classification boundaries, however, some can be bad boundaries as the classification is done poorly.

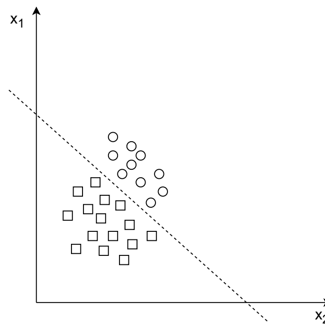

Figure 2.1: Example of a linear classifier [16]

Because single-layer perceptrons have a linear classification, it limits their potential for linearly separable patterns. This happens because they have only two layers inside the neural network. Thus, this type of perceptron can only be used in simple classification problems. This limitation came to an end when multi-layer perceptrons came about.

Multi-layer Perceptron

Multi-layer perceptron neural network (MLP) is a supervised multi-layer network that has at least three layers - input, hidden and output layer [16]. These neural networks can have more than one hidden layer. Each layer has multiple perceptrons. These perceptrons are not connected to each other in the same layer. They only connect to the perceptrons on their right layer, making all layers fully connected. A perceptron is a type of neuron that emulates a real biological neuron.

Every node, except for the ones in the input layer, uses a nonlinear activation function. This is a function that allows the creation of a complex neural network while having multiple layers of neurons. MLP networks also use a technique from supervised learning called backpropagation, which [37] explains to be a technique that calculates the loss function according to each node's weight. MLP networks are used for mode complex classification problems.

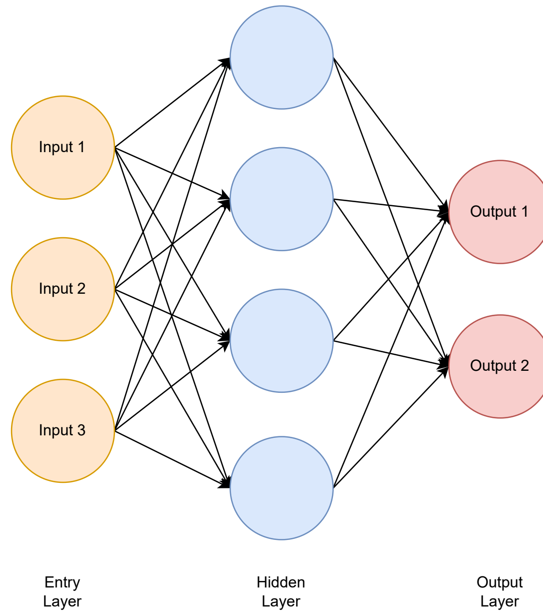

Figure 2.2: MLP Network

Radial basis function artificial neural networks

Radial basis function (RBF) networks differ from other neural networks as they have one input layer, a single hidden layer with radial basis function nodes and an output layer [33].

Inside the hidden layer is where data is transformed into a more linearly separable space and so, the number of neurons should be bigger than the ones on the input layer. The computation in the hidden layer is based on comparing the similarity between an input vector and a prototype vector. Each neuron inside the hidden layer computes this bandwidth with the following equation (from [33]):

$$\phi_i = e^{\left(-\frac{\|\bar{x}-\bar{u}_i\|^2}{2\cdot\sigma_2^i}\right)}$$

Where $\bar{x}$ is the input vector, $\bar{u}$ is the ith prototype vector, σ is the ith bandwidth and ϕ is the ith neuron output.

In the output layer, there is an activation function which performs the computation between the input vector and its weight. In the end, the resulting outcome can be used for classification problems as well as regression tasks.

2.4.2 Recurrent neural networks

Recurrent neural networks (RNNs) have connections between nodes that can form directed (nodes are connected by directed edges/arrows) or undirected graphs (there is no direction between the connections) [27]. Because of this, RNNs can use their memory to use previous information from prior inputs to influence the current ones. These neural networks are similar to multi-layer networks, but this time, data loops around the network for better predictions. It uses sequential or time series data. There are many types of RNNs with different architectures:

- **One-to-one** - Has a single input and a single output;
- **One-to-many** - Has a single input and multiple outputs;
- **Many-to-many** - Has multiple inputs and multiple outputs. Takes a sequence of inputs and generates a sequente of outputs;
- **Many-to-one** - Has multiple inputs and generates a single output.

These types of neural networks can be used in complex problems such as text and speech data, but their computation can be very slow. It also can lose weight information the deeper the network gets.

2.4.3 Convolution Neural Network

Convolution Neural Networks (CNNs) are used for visual imagery and video processing projects [26]. CNN's are versions of MLPs, so each neuron in one layer is fully connected to all neurons from the next layer. CNN's take advantage of hierarchical patterns in data and use simpler patterns when they are complex, thus reducing over-fitting data. These neural networks extract prominent features from the input using the convolution operation, a linear operation where weights are multiplied with the input images.

2.4.4 Comparison between CNNs, RNNs and MLPs

The three main categories of neural networks are convolution neural networks, recurrent neural networks and artificial neural networks. As multi-layer perceptrons are part of the ANNs category, and since we will use them in this work, we will compare them with the other two main categories.

As we can see in table 2.4, each neural network has its own advantages and disadvantages that should be considered when applied. This is an overview of the main categories in neural networks, so their applications could be extended if we dive deeper into their subcategories. All in all, there are solutions for every type of problem when using neural networks.

Despite the greatness that neural networks can achieve, there are some downsides which can deviate one's opinion about using them in their projects. Neural networks generate models that are, more often than not, complex for humans to understand their outcomes and the reasoning behind a decision. Another known disadvantage is the type of data that is used for model training. Model accuracy is outlined by the quantity and quality of training data. It is also useful for this data to not have faults so that the neural network is trained correctly.

Table 2.4: MLPs vs RNNs vs CNNs

	MLP	**RNNs**	**CNNs**
Data	Tabular data, text data	Sequence data	Image data
Recurrent connections	No	Yes	No
Parameter sharing	No	Yes	Yes
Spatial relationship	No	No	Yes
Fixed Length Input	Yes	No	Yes
Performance	Least powerful	Less feature compatibility compared to CNNs	Most powerful
Application	Facial recognition and computer vision	Text-to-speech	Facial recognition, text digitization and natural language processing
Advantages	Fault tolerance, able to work with incomplete knowledge	High accuracy in image recognition problems	Remembers every information, time series prediction
Disadvantages	Hardware dependence, unexplained behaviour of the network	Large amounts of data needed for training	Gradient vanishing

2.5 Technologies

In this section, we will explore our options in technology that allowed us to complete this project. Herein, we will discuss in detail which technologies we used and why we chose them.

2.5.1 Type of neural network

ANNs have been used in WSNs systems for failure detection [34] and forecasting [12, 41]. Similarly to this project, neural networks are responsible for predicting future values in an aquatic environment and detecting anomalies. Despite using different types of ANNs, all three projects were able to detect anomalies and make forecasts for needed values. The authors affirm that big chunks of data were used to train models in order to achieve better accuracy.

In the case of this thesis, our objective is to forecast values for a specific period in time according to previously received measurements. So, in order to do that, we need a neural network that is able to process big blocks of data for training and learn the normal behaviour of an aquatic environment so that it can accurately forecast a measurement. Anomaly detection will be done separately from the measurement forecast, so it will not be considered in this section. For continuity reasons, our work continues with the same approach as our predecessors in [21] and [37], by using multi-layer neural networks to calculate forecasts and the respective training of models.

This type of neural network has at least three layers that can be completely configured by us, developers, as well as the user. For this work, [21] explains that ANNs require specifications about the number of hidden layers and neurons in each layer. As the more hidden layers we have the worse the performance is, we are left with the decision of using one or two layers. Also, we will not use complex data, so it does not justify the use of more than two layers. By using one layer, we can approximate any continuous function for inputs within a specific range whereas using two hidden layers, we can approximate any continuous function with any accuracy which showed to be the best option.

As for the number of neurons, we will use the same number as before, 20 neurons for the first hidden layer and 15 for the second one. This neural network size showed to be the best according to tests made by [21] which showed that this option was the one with the best performance.

2.5.2 Implementation technologies

As this work is an extension of the previous work from [37], we did not change the programming language of the framework. However, its architecture did change in order to follow basic Python programming concepts and rules. Other technologies were used to further expand the framework capabilities. Our system as a whole was going to have multiple services and so, an architecture using microservices was the best approach.

Docker

The usage of multiple software for the platform that would be running all at the same time meant that we had two options, either use containers or virtual machines [22]. Using containers as opposed to virtual machines has its advantages such as containers are lightweight, their performance is not limited, it is quicker and requires less memory space.

Docker is a platform that can hold multiple software packages called containers. Containers are isolated from each other but can communicate with one another. In each container, we can install the software needed for a specific task and configure their involvement in the system.

We chose Docker for this project as opposed to other technologies like Kubernetes because it is simpler to develop and deploy. It is fast and the configuration is simple. Due to the size of this project, it also did not make sense to use Kubernetes, as it is used in larger and more complex projects.

Docker has some key concepts that should be known when working with this technology. A Docker Swarm environment is a tool for scheduling and clustering containers that provide tools for security, scalability and networks for our applications. It is a cluster of physical or virtual machines that is controlled by the swarm manager. Docker daemon manages containers on a single host. It listens for REST API requests and according to them performs container operations. Containers are a virtualized run-time environment where software is allocated. Containers have a standardization feature which allows the software to run in a similar environment with identical circumstances and it also simplifies data sharing with other containers. On the other hand, images are unchangeable files containing the source code needed to run the application in the container. Images work as snapshots of the application at specific points in time, which is useful to test different solutions. One container can have multiple layers of images and are dependent on images to run. Docker images are published and stored in an infrastructure

called image registry. It can be private or public like the well known Docker Hub [1], which has already many certified and official Docker images ready to use.

Grafana

For data visualization, we considered two technologies, Grafana [3] and Kibana [5].

Grafana is an open-source software that generates dashboards which are connected to a database. In order to visualize data, we query that database. Grafana offers various different graphical visualizations of metrics. It offers different configurations which are helpful for developers as it allows a custom setup with the technologies already in use. It supports various databases, such as Graphite, Prometheus, InfluxDB, and MySQL, among others.

Kibana is also an open-source software that uses ElasticSearch [2], a search engine. It is more focused on logs rather than visualization of metrics. Kibana only uses ElasticSearch data sources, limiting our choices. However, it has a built-in anomaly detection system which would be perfect for this work.

When we analyzed these pros and cons, we decided to choose Grafana for this project, due to the fact of it being versatile with databases and configurations. As we would be using many different technologies, it was important that the software would be compatible with them. Due to Kibana's limitations, we felt that the best decision was Grafana.

2.6 AQUAMON

AQUAMON [18] has the objective to develop a dependable platform based on WSNs to monitor aquatic environments. This project offers mechanisms to deliver a timely notification to managers and/or authorities and users of the system with the help of real time monitoring coming from WSNs.

Communication between nodes and the gateway was at first thought to be secured by LoRaWAN [13]. However, it was then discovered that LoRa shows several limitations when applied to aquatic areas as well as when the nodes antennas are placed at low heights. To address this issue and since LoRaWAN protocol is not suitable for node-to-node communication, J. Cecílio in [19] adapted the protocol and created a new one called AQUAMesh that supports communication between mesh networks. The new protocol supports multi-hop networks to allow the nodes outside the communication range of the gateway to communicate between nodes so that the gateway receives the message. This way, AQUAMesh provides communication in a wide monitoring area such as the Seixal bay and allows to deliver data for further processing, analysis and visualization in servers.

This thesis is a continuation of the work from J. Penim [37], where the ANNODE framework was implemented in an online environment. This framework was developed to receive data, and process it by detecting outliers and replacing those values with forecasts according to previously received data. In J. Penim's work, neural networks were used to predict measurements based on entry vectors composed of received data. Data from the Center for Coastal Margin Observation & Prediction [9] was used to emulate data that would be collected in the Seixal Bay.

The main types of errors and faults that the framework can reportedly detect are:

- **Outliers** - Measurement that is different from the others;

- **Drifts** - Difference between the observed value and another value;
- **Offsets** - Data that is highly displaced from the real value;
- **Noise** - Corrupted data.

Despite the detection of four different anomalies, the source code we started from only detected outliers. This source code was implemented by J. Penim in [37].

Prediction models were created with artificial neural networks that explore measurements and detect anomalies to identify these errors and faults. When an anomaly is detected, the framework offers capabilities to correct the value that originates the fault. It is also calculated a Cumulative Density Function of the Probability of Quadratic Errors that is necessary to detect outliers and estimate the quality of a measurement.

The work developed by J. Penim [37] is an extension of the first version of the ANNODE solution [21]. The first version of the framework is able to find and correct outliers. However, it was designed to run in an offline environment. Thus, an online logic was implemented where the framework receives data in real time and it is capable of detecting and correcting anomalies almost instantly.

2.7 Summary

This chapter exposes the current state-of-the-art in data and sensor data quality. We pointed out the main types of errors that occur in data collected by sensors and how we can detect and correct them. An important method to note is the use of artificial neural networks. This is the method is explored throughout this project and thesis.

Some research was made on past projects with the same context. All of them had one thing in common, the lack of a platform to process data and have reliable results. Previous projects from LNEC were explored that ultimately lead to this investigation.

For a better understanding of how neural networks work and how they can be categorized, we presented three main neural network architectures and explores how they work and how to choose the most suitable type considering the project at hand. In the end, we present a comparison between the three architectures.

We also present an introduction to the technologies that we used in this project such as Docker and Grafana. As other options were considered during implementation, we offered the reasoning behind our decision why we did not go for other suitable technologies.

The chapter is finalized with a brief introduction of the AQUAMON project, detailing small additions that were made throughout the years. As this is a continuation of another thesis, we expose the last version of the framework with a presentation of what it was capable of.

Chapter 3

Proposal

The chapter aims to explore the different methods that were planned for this work as well as the ones that were already used in the previous framework version. In this chapter, we will also present the proposed platform for data quality in aquatic environments.

The previous version of the framework supports the detection of outliers and missing data. It has services running simultaneously which waits for measurements to execute their purposes. ANNODE also advocates for real time processing of data, forecasting values and replacing them in run-time. Data from the Center for Coastal Margin Observation & Prediction [9] was used to consider a new environment. To emulate data that would be collected in real time in the Seixal Bay, we used data from LNEC's sensors that were fetched from their API. The goal for this project is to re-implement the frameworks architecture, improving its flow and understanding for future new iterations; create Docker containers for the framework and all its services; implement a new dashboard and an API to fetch all types of data that the framework produces. Finally, we aim for a framework that can receive any time-series dataset containing one or more data sources and variables. In this context we re-design ANNODE, creating a new version ANNODE+.

3.1 System Architecture

The system architecture is represented in Figure 3.1 with all the necessary components for the implementation of the project. Like many system architectures, ours is divided into three layers, from top to bottom, the computation layer, the application layer and the interface layer. The computation layer is where we find a Docker environment responsible for all Docker containers and maintenance of the system. The computation layer is the system's core that gets everything running and supports all features of this project. The user does not have access to this layer, as it only maintains execution through containers. By using Docker we ensure that containers are isolated from each other. Using Docker containers also help with storing code in Docker images, thus creating a lightweight environment with an available and capable platform. To create images we will be using Dockerfiles for custom images and certified images from Docker Hub.

The application layer is where we have the docker containers necessary for the execution of the framework and dashboard. Here, we have two containers for database control, a MySQL container and a PHPMyAdmin container which allows developers to have a better view of the MySQL database. An

API container is also present where the user can interact by making requests for the framework's results. Sensors also interact with this layer when they send measurements to a channel that is being listened to by the framework's container.

Finally, the interface layer is where it is provided communication between the user and the system through a mobile and desktop dashboard.

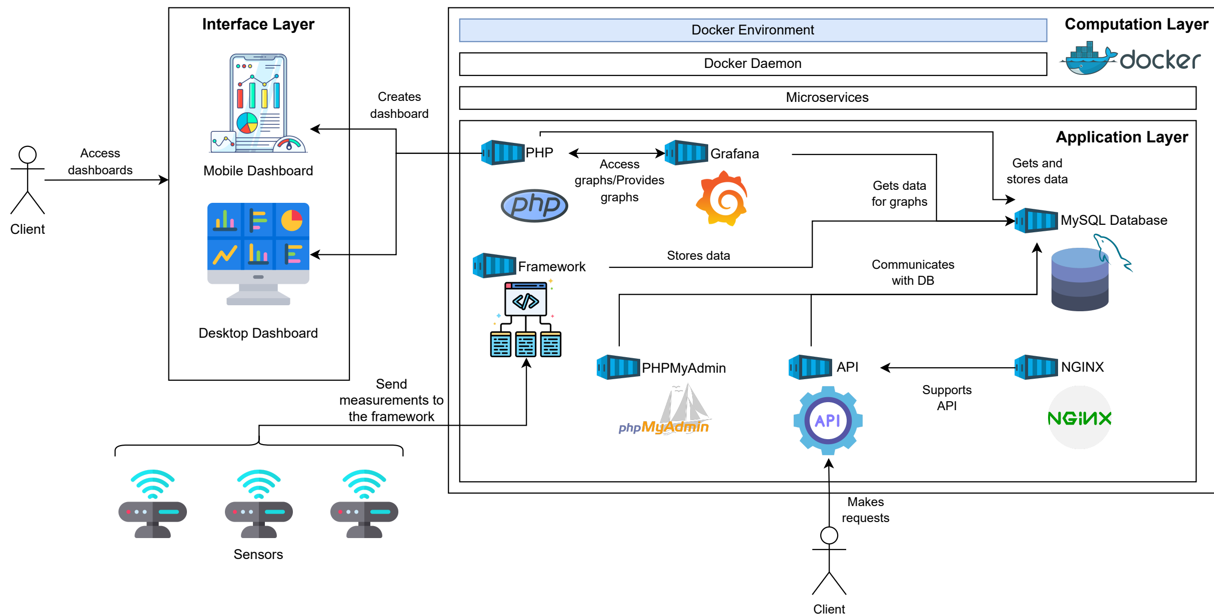

Figure 3.1: System architecture

3.2 ANNODE+ Architecture

ANNODE+ is split into two parts, one for training neural networks (Training Block) and the other part is a real time handler that processes data and detects possible anomalies (Execution Block), in this case, outliers and missing data. This type of architecture was inherited by the original framework and would not be changed.

Figure 3.2 shows the main flow of the framework. The offline block is the Training Block whereas the online block is the Execution Block. The Training Block contains the trained neural networks and the Execution Block is where the framework works. In here, we typically have five steps:

1. The sensors sends collected data/measurements through the network to the framework;

2. Data is received and stored by the framework;

3. The framework checks if there is any omissions on the communication between the sensor and the server. If an omission is detected, it is processed accordingly so that the framework can later replace the value for the respective prediction;

4. When no more omissions are detected, the framework starts calculating forecasts with the trained neural networks from the offline block;

5. Once we have the forecast, the measurement received goes through a quality assurance where it will determine if that measurement is an outlier or not. This process will be explained in detail in

Sections 3.2.2 and 4.3.1. If it is an outlier, the received measurement is replaced by the forecast calculated in step 4.

If we have more than one sensor available at the time of training, each sensor can have up to three different neural networks:

- A neural network that represents the target sensor, with data **from the same sensor** whose measurement is being processed (temporal correlation);
- A neural network that represents the target sensor, with data **from the same sensor and its neighbors** (temporal and spatial correlation);
- A neural network that represents the target sensor, with data **from its neighbors** (spatial correlation only).

Having three different neural networks will be useful to have some redundancy when it comes to multiple sensors which can be affected by physical events and interfere with the collected data. This process is done in the quality assurance step (step 5).

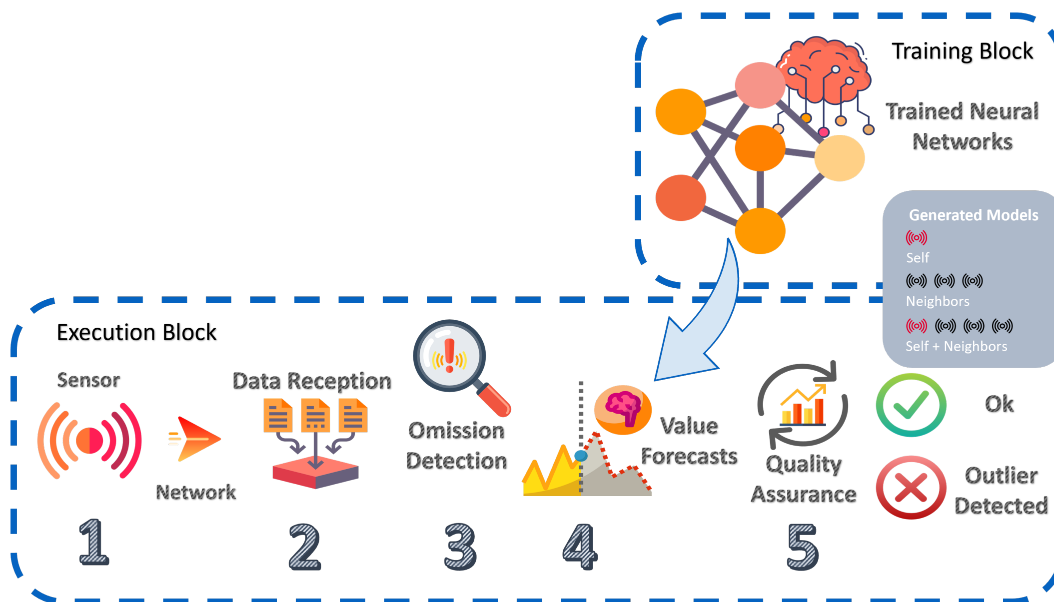

Figure 3.2: Framework's overview

3.2.1 Training Block

In the Training Block, neural networks are used to train models that later help on detecting faulty measurements. This training step is supported by a dataset containing sufficient information to represent all the main characteristics and dynamics of the variable being modelled (e.g., represent the seasonality present in the real phenomenon). The user must prepare a configuration file with all the training requirements, such as the number of sensors, their characteristics (e.g., sampling period), and data characteristics (e.g., representative period). This file also contains specifications on how many neurons and layers the neural networks should have as well as how training will be executed.

When the configuration file is sent to the framework, data is first processed in order to create a CDF (Cumulative Distribution Function) file. We should now have three datasets, the CDF file, a training

dataset (usually 70% of the original dataset) and a dataset for testing (usually 30% of the original dataset), as shown in Figure 3.3. As data that come from sensors are most likely not to be received at the same time, during this process, data needs to be aligned so that we can create entry vectors correctly to help with the calculation of the CDF. These two mechanisms will be explained in section 3.2.1. At the same time, non-trained neural networks are created to later receive the processed data and be trained.

Models are trained using datasets that must be provided by the user, which must have been previously collected and must only include data that is considered correct. Annotating or cleaning training data is a typical requirement when using ML methods. This is why the training dataset should not have anomalies.

When data from multiple correlated sensors is available, several models need to be created, corresponding to different combinations of sensors. This is because the methodology ANNODE there are three different types of neural network models. In these cases, we consider three options: *all* (considers data from all sensors), *neighbours* (only considers data from the main sensor's neighbours) and *self* (only considers data from the main sensor). These models are created separately and the user chooses which to train in the configuration file. In fact, to predict the next measurement of a sensor, it is possible to use a model that was trained using only data from that sensor (exploiting temporal correlation) or using a model combining data from multiple neighbour sensors (exploiting spatial correlation).

As soon as the neural networks are trained, we use the most recent model for the calculation of the CDF. CDFs are calculated using the quadratic error between a prediction and a value. Finally, the neural networks are stored locally to be used in the Execution Block.

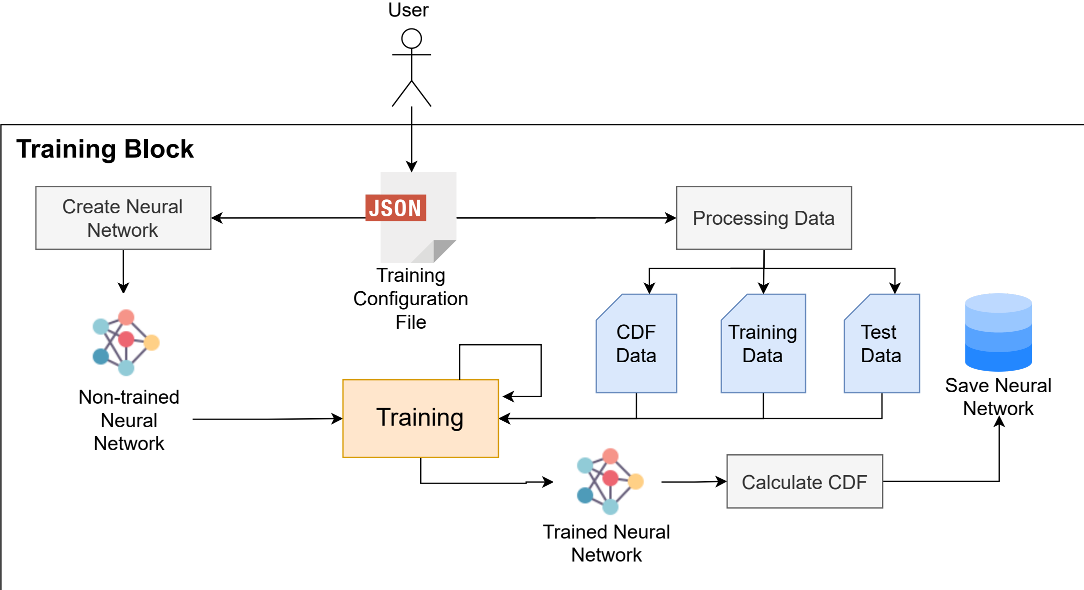

Figure 3.3: Training block architecture

Data processing configuration file

A configuration file needs to be sent to the framework with all the parameters the user wishes for how training is going to be performed. This file needs to be in accordance with other configuration files that will be explained in the following sections for the framework to be able to process data with the same parameters as the training ones.

```
aquaIoT - configuration_training.json

28  "training_sets": {
29          "raw_path": "data/raw_main/",
30          "save_path": "data/processed/",
31          "metrics": ["temp"],
32          "n_sensors": 4,
33          "run_periods_self": 60,
34          "run_periods_others": 20,
35          "period_time": 120,
36          "skip_period": 5
37      }
```

Figure 3.4: Training sets configuration file

Figure 3.4 represents part of the training configuration file which is dedicated only to the creation of training data explained in section 4.1. It has eight parameters:

- **`raw_path`** - directory path where raw data is located;
- **`save_path`** - directory path where processed data will be saved;
- **`metrics`** - list of metrics that will be trained. These metrics need to be the same as the sub-directories of `raw_path`;
- **`n_sensors`** - number of sensors available for training;
- **`run_periods_self`** - number of measurements from the target sensor considered for entry vector creation within a full tide;
- **`run_periods_others`** - number of measurements from neighbour sensors considered for entry vector creation within a full tide;
- **`period_time`** - period of time in seconds between each measurement;
- **`skip_period`** - minimum tax of sampling.

The parameters `run_periods_self` and `run_periods_others` will determine the entry vector's length. In this case, we have four sensors, the target sensor will provide 60 measurements to the entry vector while the other 3 sensors will provide 20 measurements each. Thus, the entry vector's length will be 120 measurements. The framework will receive new measurements every 120 seconds and it will use measurements, spaced within themselves every 5 minutes, for processing.

Training configuration file

As part of a bigger configuration file, we have the configurations that will regulate training. Here the user will indicate which model is going to be trained, sensor information and configurations for the neural network. This file should be filled out completely and correctly for a good training process.

```
aquaIoT - configuration_training.json

2  "training": {
3          "data_train": 0.8,
4          "data_test": 0.2,
5          "epochs": 2000,
6          "hidden_layer_1": 20,
7          "hidden_layer_2": 15,
8          "output_layer": 1,
9          "loss_function": "mean_squared_error",
10         "sensor": "tansy",
11         "metric": "temp",
12         "inputs": "all",
13         "id": 1,
14         "n_other_sensors": 3,
15         "input_shape": 120,
16         "input_target": 60,
17         "input_others": 20,
18         "checkpoint_epochs": 100,
19         "checkpoint_path": "ann/training/",
20         "model_save_path": "ann/models/",
21         "data_path": [
22             "data/processed/21jul_2009-18oct_2009/temp/tansy.npz",
23             "data/processed/11nov_2009-05jun_2010/temp/tansy.npz"
24         ],
25         "cdf_data_path": "data/processed/20aug_2010-10oct_2010/temp/tansy.npz",
26         "runs": 10
27     }
```

Figure 3.5: Training configuration file

Some of the most important parameters are:

- **`data_train`** - Percentage of data that will be dedicated to training the model;
- **`data_test`** - Percentage of data that will be dedicated to testing the trained model;
- **`epochs`** - Number of times a network passes backwards and forwards;
- **`hidden_layer_1`** and **`hidden_layer_2`** - Based on previous work in [21], the best number of hidden layers is two. The first hidden layer has 20 neurons and the second one has 15 neurons;
- **`output_layer`** - As this is a neural network that will give as an output one value, we only need one neuron in the last neural network layer;
- **`loss_function`** - Name of the loss function desired to use;
- **`sensor`** - Sensor's name;
- **`metric`** - Metric that corresponds to the dataset's measurements;
- **`inputs`** - Has three different options: *all*, *neighbours* and *self*. Corresponds to the type of model that will be trained;
- **`n_other_sensors`** - Number of sensors without counting the target sensor;
- **`input_shape`** - Total number of measurements considered for training for each sensor;
- **`input_target`** - Total number of measurements considered for training from the target sensor;

- **`input_others`** - Total number of measurements considered for training from the rest of the sensors;
- **`runs`** - Number of neural networks that will be trained.

Entry vector's creation

To train neural networks and compare measurements to models, the framework uses entry vectors. Entry vectors are lists of values used not only to train neural networks but also to make predictions. Before we can start model training, the data needs to be processed in a procedure where 70% is dedicated to training and 30% is to test trained models. During this process, entry vectors are created according to an approach that is chosen by the user in the training configuration file. There are three different approaches available:

- **Exponential approach** - an original approach where the algorithm would exponentially fetch a measurement from the previous tide;
- **Linear approach** - the algorithm linearly fetches data from the previous tide;
- **Last ten approach** - the algorithm fetches the last ten received measurements.

These vectors are created based on the values that are received. The values we receive from multiple sensors need to be aligned to create correct entry vectors. Otherwise, it could cause problems in the execution of the framework. For instance, in an example of four sensors, the target sensor starts its measurements at 09:00 a.m, and its neighbours start their measurements at 09:10, 09:15 and 09:30. The values can only be aligned after 09:30 because that is the only time the four sensors have the same number of measurements in common. By aligning the times, the timestamps of these measurements will be rewritten to 09:30 a.m. The alignment of times will be explained in more detail in section 3.2.1.

After the measurements' alignment, the entry vectors can be safely deployed. Its creation uses a given number of measurements. The user in a configuration file chooses this number.

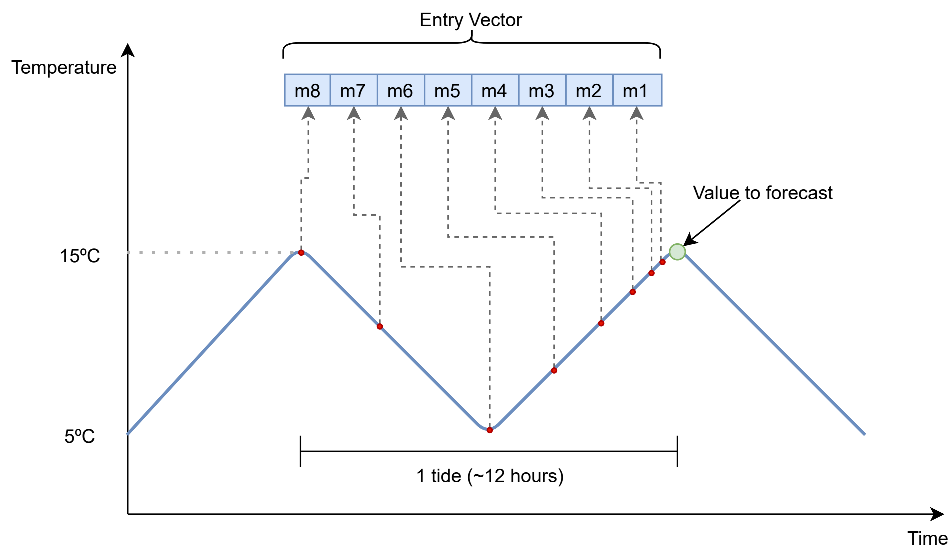

Figure 3.6: Example of the entry vector's construction using the target sensor in an exponential approach [37]

As seen in Figure 3.10, the entry vector's construction is made by using previous measurements that are chosen in an exponential approach. In other words, it is first calculated how many measurements there are in a tide period (twelve hours). That number sets a maximum index which cannot be surpassed. It is then exponentially determined the chosen N indexes from zero to the maximum index number. Finally, the entry vector is completed with the values from each chosen index, making it a vector with data equivalent to the last twelve hours of received measurements.

In the linear approach, the entry vector will be filled with values spaced between themselves linearly. The entry vector should consist of values corresponding to twelve hours of measurements. In Figure 3.7 we have a value to predict in green, where the entry vector is complete with values equally spaced.

In the last ten approach, the entry vector is completed with the previous ten measurements. These values are the ones immediately before the value to predict. Albeit the entry vector does not have enough measurements to represent a full tide, the last ten measurements will most likely be very similar to the value to forecast. In Figure 3.7 there is an example of how this algorithm works.

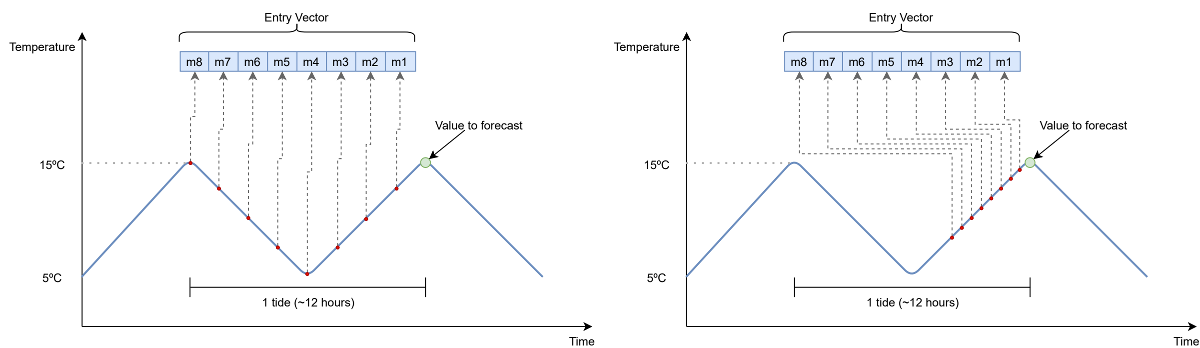

Figure 3.7: Linear approach (left) and last ten approach (right)

Align times

Measurements need to be aligned in a network of multiple sensors. As explained in the previous section, sensors do not send data at the same time, thus resulting in dispersed timestamps. These timestamps should be aligned for the framework to successfully create entry vectors. Due to missing data and delays during communication, measurements are not received and aligned with other sensors. This is explained in detail [37] in figure 3.8. It demonstrates a perfect example and a realistic example, where we have three sensors, the blue represents the target sensor and the orange represents its neighbours. As we can see, in a perfect example, every measurement sent from *sensor neighbour2* has the same times as all measurements from the target sensor. Whereas in the realistic example, there are some delays and missing data in the *sensor neighbour1*'s measurements times. This is what happens in real aquatic environments, so data that goes through the framework must be aligned.

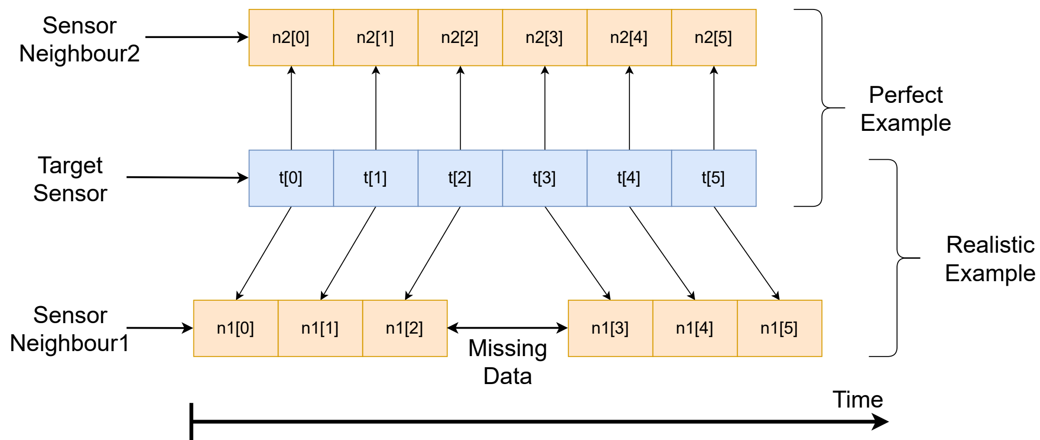

Figure 3.8: Alignment of Times [37]

Training neural networks

Once our data is processed, we can start training models. First, the user fills out a configuration form where the neural networks' shape is delimited, as well as important information such as which sensor data is being trained and its metric. Then, data is processed as explained in section 3.2.1. Processed data is the type of data that is used to train models. Depending on the number of runs the user chooses in the configuration file, n neural networks are trained. The neural network with the least loss, i.e. the one with the best performance, is the chosen one to calculate forecasts later in real time data processing. MLP networks use a backpropagation algorithm which adjusts the weights and biases from each neuron to lower the final value of the loss function [37, 41]. As weights and biases from each neuron is assigned randomly by the framework, each trained model has different loss values. The best model is found using cross-validation comparing loss values with other trained models for the same sensor. This process is repeated for each one of the three different models of each sensor, explained in section 3.2. In the end, we have trained models and a Cumulative Density Function (CDF) used during processing for anomaly detection and quality assurance which is represented in Figure 3.9. The Cumulative Density Function is calculated by considering the log-logistic function of each square errors distribution for all sensors and respective predictions.

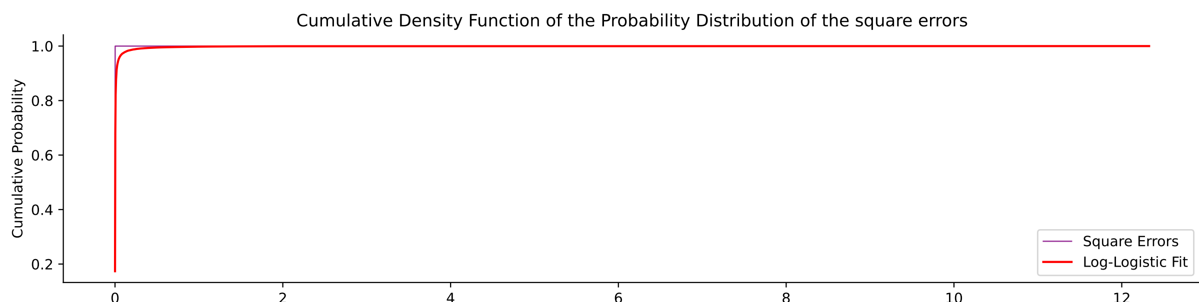

Figure 3.9: Cumulative Density Function of the distribution probability of the square errors

3.2.2 Execution Block

In the Execution Block, we propose a two-service implementation approach: Communication Service and Processing Service. The Communication service is responsible for receiving sensor data, identifying its source sensor, and for inserting received data in a sensor measurements queue, which is used to communicate with the Processing Service. This queue only receives one measurement at a time and is unitary.

The Processing service is triggered by new incoming measurements. Before processing begins, the framework detects any possible omissions between the last stored measurement and the latest received measurement. Each sensor s sends a new measurement every P_s hours/minutes/seconds. If the last measurement is received after $P_s + \delta$, where δ corresponds to the jitter assumed for the measurement reception instant, an omission is detected and a special value (NaN) is inserted in the queue. A jitter is a variation in time delay in a communication from when a signal is transmitted to when it is received. By considering it, the probability of wrongly detecting an omission is reduced to the probability of considering a wrong (too small) jitter. However, actual processing only starts after a certain number of measurements have been received, covering an entire representative period of the physical process being monitored (e.g., 12 hours for sea water level). When this condition is fulfilled, the actual processing is executed for every new incoming measurement.

Firstly, measurements are temporally aligned, to match the alignments used during model training. Then, input vectors are built from the stored and aligned measurements, to be fed to the relevant ANN prediction models. If correlated sensors are available, the three models previously explained in section 3.2 are used to generate forecasts.

The benefit of using these multiple forecasts is to be able to distinguish real environmental events (even if they look like outliers) from real outliers (only affecting the target sensor, but not the neighbour ones). It is then possible to check the correctness of the received measurement, comparing it with the generated forecasts. In fact, given that sensor data can be affected by different kinds of errors, it is at this point of the processing chain that it is possible to determine if some failure may have happened, leading to those different errors. While ANNODE+ is only detecting outliers and missing data, it may be extended in future work with new failure detection blocks to detect data drifts and noise. When some failure is detected (either missing data or outlier), then the received measurement (or the special NaN value) is replaced by an average of the calculated forecasts.

In addition to detecting failures, it is also possible to calculate the quality of the received measurement. To calculate the error between forecasts and measurements, the framework applies the squared prediction error formula, which is useful for the Cumulative Density Function. When the difference between the received measurement and the forecasts provided by the models is small, then the quality is high. Failure detection and quality assurance are part of a Prediction and Quality Service.

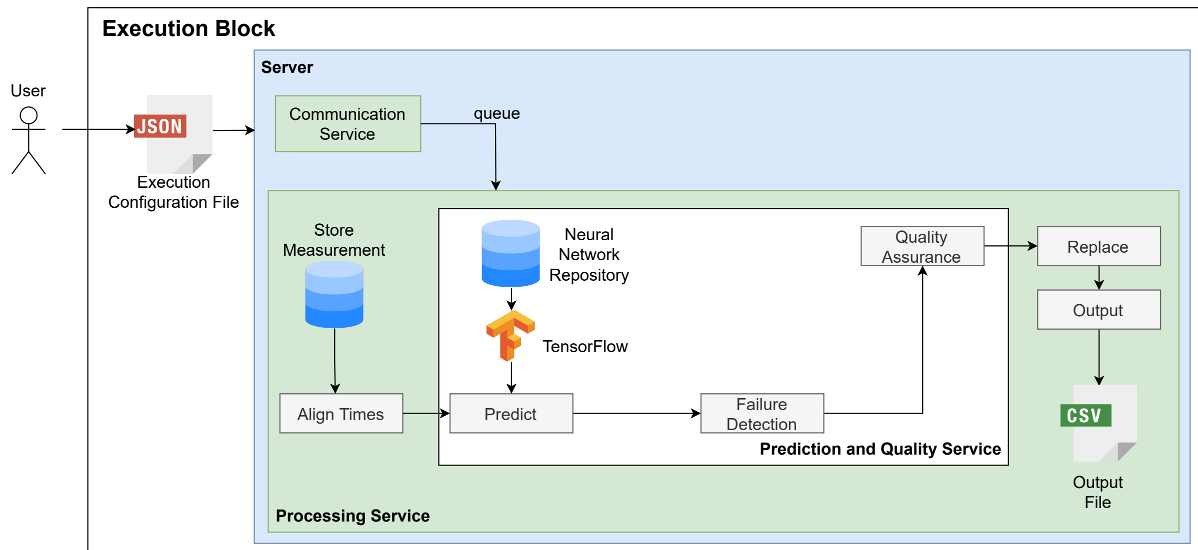

Figure 3.10: Execution block architecture

Execution configuration file

To have more control over the framework when processing data, a configuration file was created so that we could change some specifications about the execution of the framework, such as the period each measurement is set to arrive and sensor information. The framework will run according to this configuration file and if it is not in hand with the previously trained models an error is shown to the user. This file is divided into three sections, one that gives information to the framework on how to execute, the other a section for communication parameters and finally, a section for sensor data.

master_thesis - configuration_processing.json

```
{
    "sensor_handler": {
        "run_periods_self": 10,
        "run_periods_others": 0,
        "approach": 1,
        "cdf_threshold": 0.9985,
        "skip_period": 0,
        "period_time": 900
    },
    "communication": {
        "host": "",
        "port": 9999
    },
    "sensors" : [{
        "sensor_id": "lnec",
        "path": "data\\raw_other\\lnec\\temp\\lnec_data_set01_set08.csv",
        "data_type": "temp",
        "latitude": 38.644858,
        "longitude": -9.1046928
    }]
}
```

Figure 3.11: Execution configuration file

`cdf_threashold` is the significance value that will be used to determine if a measurement is an

anomaly or not, with the help of the CDF function calculated during training. The parameter `approach` receives the type of approach the user wants to use. It has three options: 1 - exponential approach, 2 - linear approach and 3 - last ten approach.

Communication with the framework

To simulate a sensor sending data to the framework, we have a script that receives the relative path of a JSON file with a dataset of measurements from a real sensor located in Seixal bay and sends data one by one every t seconds, where t is determined by the user. The communication between this simulator and the framework is done by a TCP server in which the user designates the port and the address where a connection will be created. Before entering inside the communication service, the framework inserts new sensors in the database for future reference for the dashboard.

Communication service

The communication service is simple and responsible for acquiring received measurements and inserting them in a queue that is accessible by the processing service. These measurements are received in a connection socket that is always listening for new connections in a port that is given by the user.

Processing service

The processing service is where data is handled. This service is triggered when it receives a measurement, which is added to an entity that stores all measurements. Here, the framework checks if possible data omissions could occur while the sensors send their data. If an omission is detected, the initial queue is filled with a senseless value to be later dealt with in the prediction and quality block. When this condition is fulfilled, the actual processing is executed for every new incoming measurement. Firstly, measurements are temporally aligned, to match the alignments used during model training. Then, input vectors are built from the stored and aligned measurements, to be fed to the relevant ANN prediction models. If all requirements are met, meaning there are enough values, a forecast is calculated according to the entry vector created with previously received values.

If correlated sensors are available, then the following models (which have been previously trained) are used to generate forecasts:

1. A model that only uses data from the target sensor whose measurement is being processed (temporal correlation);
2. A model that uses data from the target sensor and its neighbours (temporal and spatial correlation);
3. A model that uses data only from the target sensor neighbours (spatial correlation only).

The benefit of using these multiple forecasts is to be able to distinguish real environmental events (even if they look like outliers) from real outliers (only affecting the target sensor, but not the neighbour ones). It is then possible to check the correctness of the received measurement, comparing it with the generated forecasts. All measurements go through a filter which checks if a value is erroneous or not according to some conditions, specifically the calculated squared prediction error. If the measurement

is faulty, it is replaced with the respective forecast. In the end, we have a set of correct values without anomalies.

In addition to detecting failures, it is also possible to calculate the quality of the received measurement. When the difference between the received measurement and the forecasts provided by the models is small, then we can consider that the measurement is of high quality meaning it is coherent with other received measurements.

Omission detection

The detection of omissions is the first thing that happens inside the processing service. Once a measurement is received, it is analysed and compared with the previously received measurement. The comparison is made with both measurements' timestamps. We defined a *jitter* that is added to the period in which a measurement is scheduled to be received. If no data is received within that period, an omission is detected and filled with a senseless value to be later replaced by its prediction in that timestamp.

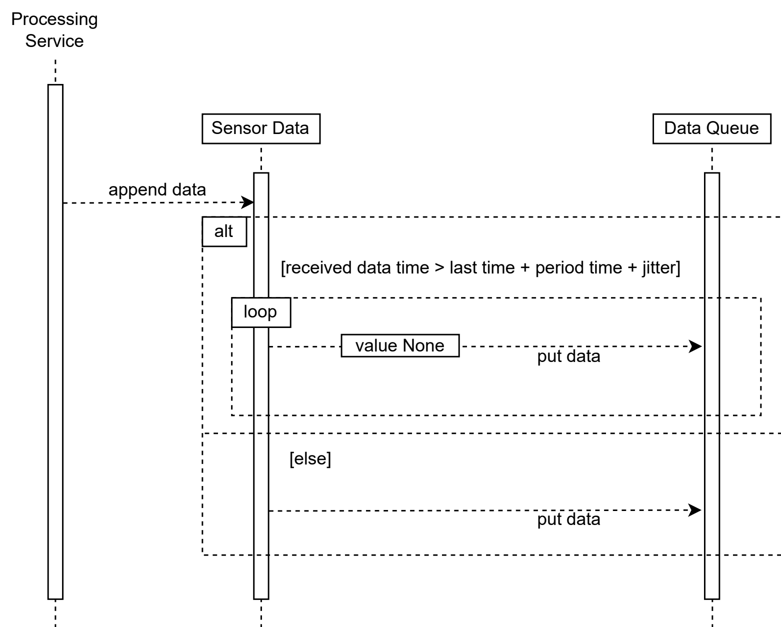

Figure 3.12: Omission detection

Prediction block

The prediction block is a module where a forecast is calculated. To calculate a prediction, the framework obtains the correct model for each sensor. In case there is only one sensor sending data, we will have a model represented by *self*, which represents the target sensor. If there are multiple sensors, then we will have three different models designated by *self* (target sensor), *neighbours* (target sensor's neighbours) and *all* (all sensors).

Finally, in order to have entry vectors with previous correct data from multiple sensors, the calculation is a bit different from the one explained in section 3.2.1. The entry vectors consist of measurements from all sensors from the last 12 hours, which represents a full tide. Figure 3.13 shows the creation of an entry vector with measurements from two sensors using the exponential approach. The algorithm uses measurements closer to the value to forecast as there are more correlations. In this case, there

are fewer measurements from the neighbour sensor in the entry vector. This happens because although measurements can be a bit different, it helps correlate measurements between sensors.

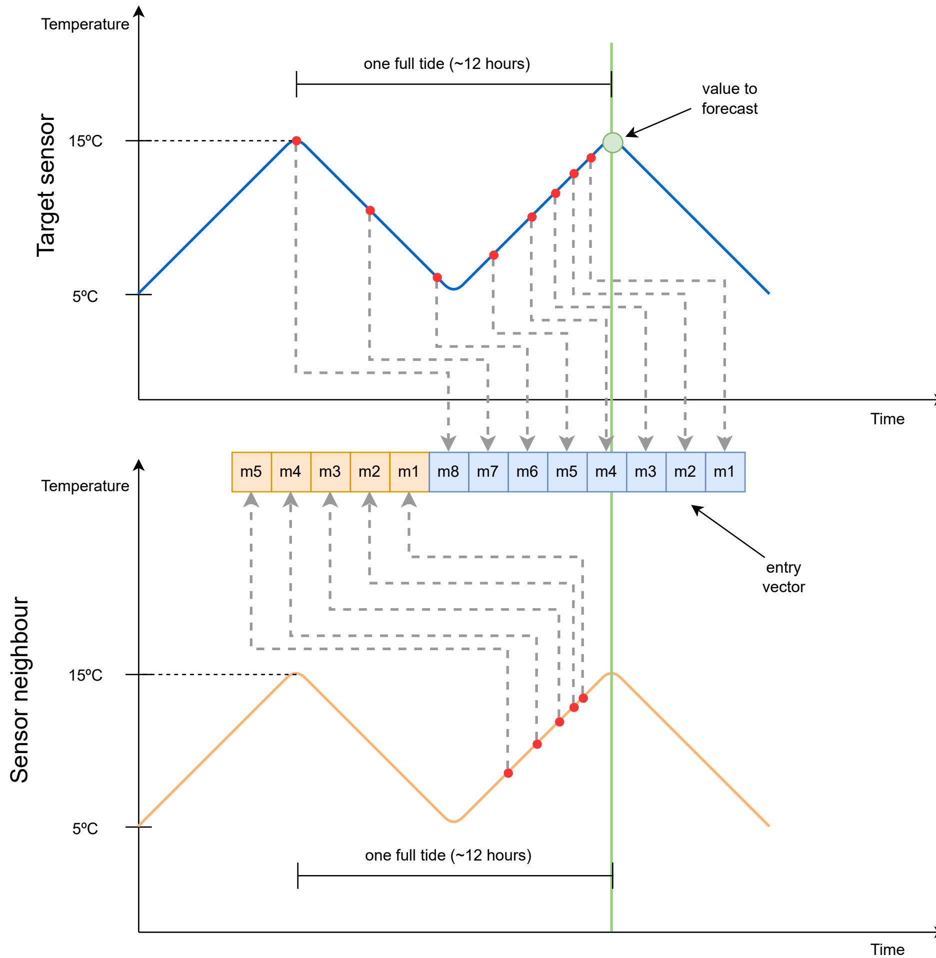

Figure 3.13: Entry vector creation with multiple sensors [37]

Quality assurance block and outlier detection

As soon we have a prediction, we enter the quality assurance block which will detect if there is an anomaly or not. Firstly, we calculate the quadratic error between the actual measurement and the forecast. The outcome value, x will be used for fault detection. During training, a CDF is created, which given x will give us the probability of a measurement to be an error.

Figure 3.14 shows how this process is done. The CDF is represented by the chart in the Figure, where the x axis is the squared error value, which can be calculated by the following formula:

$$x = \left(\frac{\sum_{i=0}^{n} e(m, p_i)}{n} \right) \tag{3.1}$$

Where m is the received measurement's value, p_i is the respective prediction and n is the number of sensors.

The y axis is the probability of a measurement to be an error. In Figure 3.14, the red dot corresponds to the probability of m to be an anomaly. Depending of the designated threshold in the configuration files by the user, if the probability (pr) is above the given threshold, then the prediction is largely different

from the received measurement.

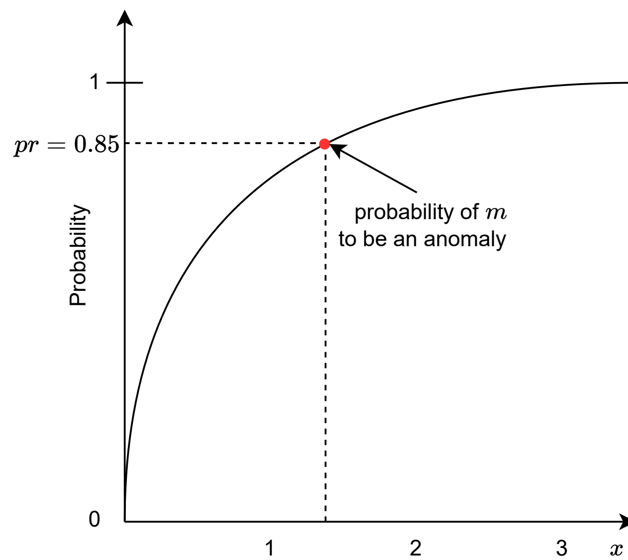

Figure 3.14: Outlier detection

If we are processing data from a network of multiple sensors, some metrics are analyzed in order to come to a conclusion about an anomaly. For instance, three forecasts are made according to the respective models (*self*, *neighbours* and *all*). Those three forecasts are compared with the received measurement, where one of three possibilities can occur:

- **Different from one prediction** - there is a high probability that the target sensor is being affected by some physical event. Not considered an outlier;
- **Different from two predictions** - if the measurement is similar to the prediction based on the target sensor, the measurement is likely correct. If the measurement is similar to the prediction based on neighbours' measurements, is possible that a physical event is forcing all measurements to take unexpected values. The case in which a measurement is similar to a prediction based on all sensors is unlikely to occur due to incompatibility with other conditions;
- **Different from all predictions** - the measurement that is not similar to any prediction is an outlier.

Of course, in an instance of only one sensor, if the measurement is vastly different from the forecast, it means that that measurement is an anomaly.

Finally, we calculate a numeric value that will quantify the quality of measurements. That value is a decimal number between 0 and 1, being 0 the worst quality and 1 being the best quality. When an outlier or an anomaly is found, the received measurement is replaced with the respective average forecast from all models available for the target sensor.

$$pr = \begin{cases} pr > threshold & 1 \\ otherwise & pr \end{cases} \tag{3.2}$$

Where *pr* is the probability of a measurement to be an anomaly.

3.2.3 Graphical Interface

As part of this thesis, the development of a graphical interface was always in the plan. To do so, we made two prototypes of what would later become our dashboard.

Figure 3.15 represents the first idea of a configuration page to control the framework's behaviour. This, of course, would be an alternative configuration for the main framework's configuration file that was explained in section 3.2.2. However, there was a need to be able to edit and/or add sensor information while in the dashboard. Thus this page would be dedicated to those features.

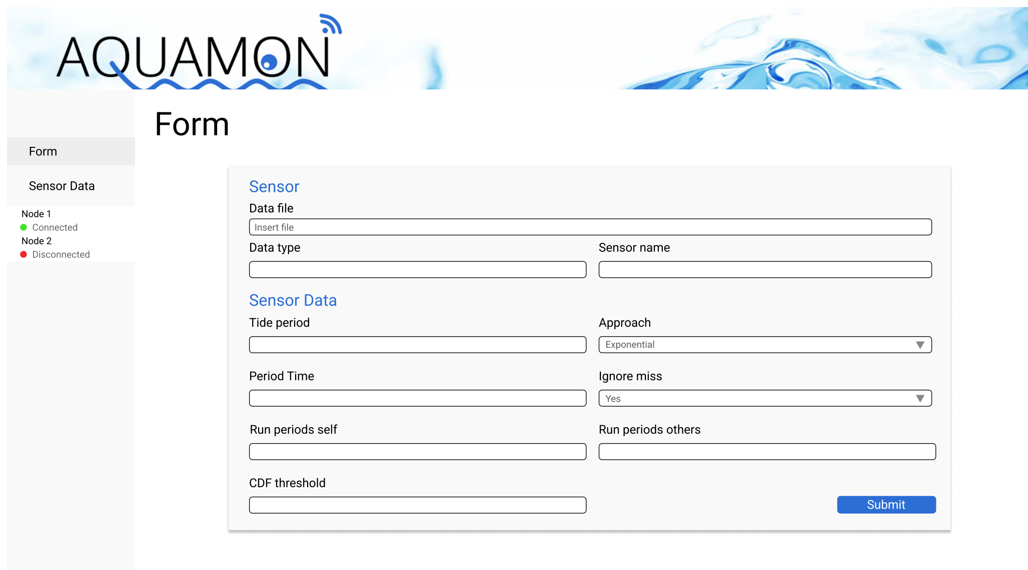

Figure 3.15: Prototype of the configuration page

For the main page, the idea was to display as much information as possible to the user. Figure 3.16 was the first idea for this page, where raw and corrected measurements were displayed in different graphs as well as a log console with every anomaly alert. Graphs would be updated automatically and in real time. The sidebar shows all nodes available and their status (Connected, Disconnected and Inactive).

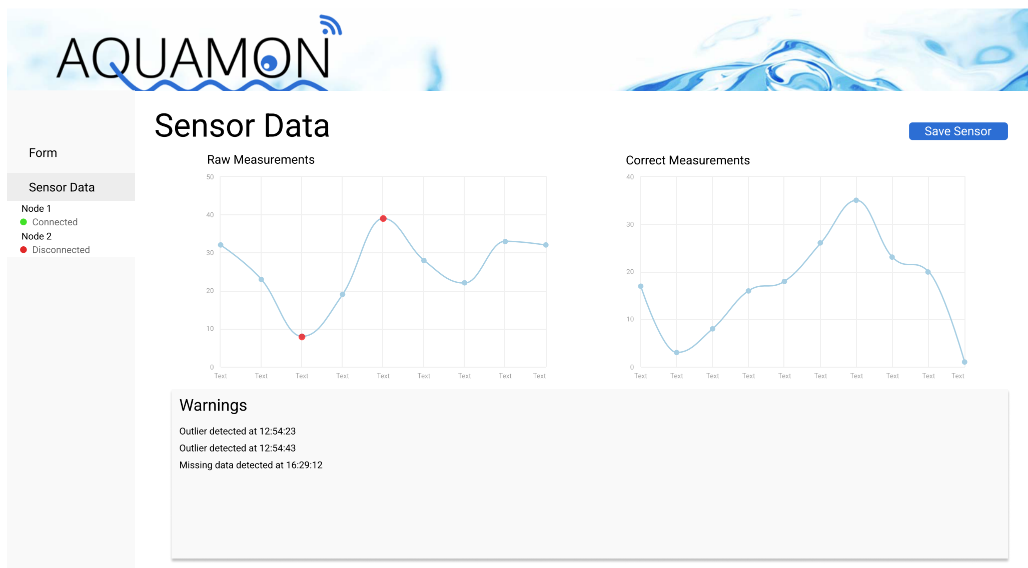

Figure 3.16: Prototype of the main page

3.3 API

In order to fetch all data that is processed by the framework, we propose an API that allows developers to get the framework's results to use in any other project they might have. This API will able to get the results from the framework. The user will be able to select the period of time that he wants to fetch and which sensor. The same goes for the type of metric. The route template is: /api/<sensor_name>/<variable>/< where <sensor_name> is the name of the sensor, <variable> is the type of metric, <begin_date is the date the user wishes to start in dd/mm/yyyy format, <period> is the number of days after *<begin_date>* and <method> is the method that will define which type of data is going to be fetched. When a request is made to the API, the result comes in a JSON file. If there is no data to be fetched, an empty JSON file will be showed. As this will be based on LNEC's API from UBEST [11], the routes are similar.

3.3.1 Documentation

<method> is the parameter that has 6 different options. They all require the previous parameters (<sensor_name>, <variable>, <begin_date, <period> and <period>) in order to work.

- **/raw_measurements** - only returns raw measurements;
- **/corrected_measurements** - only returns measurements that are correct. This includes measurements that were defined as anomalies and were replaced by their forecasts;
- **/quality** - returns the quality average;
- **/outliers** - returns all measurements that were considered to be outliers;
- **/omissions** - returns all measurements that were considered to be omissions;
- **/anomalies** - returns all anomalies (outliers and omissions).

3.4 Summary

This chapter explores five topics from the framework and system architecture to the more detailed parts of the framework that are planned to be improved. Section 3.1 gives an overview of the system's architecture. Section 3.2 explores the two parts of the framework, the training block and the execution block, detailing each of them and their components. Section 3.2.1 details new approaches for the creation of entry vectors and how neural networks are trained. Section 3.2.2 explains the two main services in the framework: communication service and processing service. For each service, specific features are detailed. Section 3.2.3 presents the prototypes of the graphical interface. Section 3.3 explores the API's documentation.

Chapter 4

Implementation

This section exposes the main techniques used by and in the ANNODE+ framework. The first phase of this thesis consisted of understanding the framework and its details, such as the data requirements, configurations and used techniques. Then, in the second phase, new features were implemented to enhance the frameworks capabilities, and a dashboard was developed using MySQL database and Grafana to help with data visualization. Finally, we used the Flask framework for the implementation of an API.

Since the framework was implemented before the beginning of this thesis, there was no need to apply new methods for detecting and correcting anomalies. The framework is based on the ANNODE architecture, which identifies and corrects anomalies in an offline environment. Neural networks are used to train models that later help detect inaccurate measurements. For the forecast calculation, a deep learning API was used called Keras [4], built on top of Tensor Flow [10]. Tensor Flow allows the creation of neural networks and statistical models. To calculate the error between predictions and measurements, the framework applies the squared prediction error formula, which is helpful for the Cumulative Density Function.

In this thesis's scope and since the framework's first version was oriented to the SATURN Observation Network data, the new version is now suitable for LNECs data from the Seixal bay and any time-series dataset. The code needed to be restructured in offline and online blocks to achieve this. This chapter will explain what was changed and how we implemented these changes in detail.

4.1 Data processing

Before starting training neural networks, data needs to be processed to be able to calculate and create a CDF file. First, the framework validates the configuration file. If all is correct, the CDF file starts to be built. The CDF file is built based on the defined parameters in the configuration file. The process generates entry vectors based on the user's chosen approach.

4.1.1 Input data structure

To make this framework more generic and available to more types of time-series datasets, we updated the function responsible for opening and reading documents sent to the framework. In the first version, the framework was specifically implemented to process data from raw files of the Saturn Observatory Network [9], which would have a very specific structure and not always the same one. Data also came

in the *.mat* file extension which is not the average file type used in these types of problems. We then decided right from the start that this framework would be able to read *.csv* and *.json* files. These files need to follow a specific structure. Otherwise, the framework will not be able to read them.

aquaIoT - dsdma_temp_1out_31dec_2013.csv

```
1  01/10/2013 00:01; 15.389
2  01/10/2013 00:07; 15.359
3  01/10/2013 00:13; 15.348
4  01/10/2013 00:19; 15.430
5  01/10/2013 00:25; 15.412
```

Figure 4.1: Example of *.csv* file structure

The *.csv* files have in the first column the date and hour the measurement was captured in datetime format (dd/MM/yyyy HH:mm), delimited by a semicolon followed by the measurement's value.

Because we use LNEC's data directly from their API, the framework reads *.json* files with their structure in mind.

aquaIoT - lnec_data_set01_set08.json

```
1  [{"name": "CC Faro", "data": [{"x": 1630454415000,
   "y": 23.95, "depth": "-"}, {"x": 1630455315000, "y"
   : 24.13, "depth": "-"}]
```

Figure 4.2: Example of *.json* file structure

These files also need to follow a directory structure, divided by metric. If we take Figure 3.4 as an example, inside the given *raw_path* data should be divided in directories of the respective metric the dataset represents as such data/raw_main/<dataset_name>/<metric>/<dataset_files>.

4.2 Neural network training

Neural network training did not suffer significant changes from the previous framework version. It contains the same methods as before with some slight changes regarding the training of the model *all*. According to the configuration file, each model has to be trained with specific parts of data, as Figure 4.6 shows. If we want to train the *self* model, we only use data from the target sensor. The same logic is used for training the *neighbours* model. We only use data from the target sensor's neighbours. Training of the *all* model has measurements from all sensors.

```
aquaIoT - ann.py
49 inputs = data[0]
50 if ann_cfg["inputs"] == "self":
51     inputs = [values[:ann_cfg["input_target"]] for values in inputs]
52 elif ann_cfg["inputs"] == "neighbours":
53     inputs = [values[ann_cfg["input_target"]:] for values in inputs]
54 elif ann_cfg["inputs"] == "all":
55     end = int(ann_cfg["input_target"]) + int(ann_cfg["input_others"]) * int(ann_cfg["n_other_sensors"])
56     input
```

Figure 4.3: Initial input for training depending on model

The three datasets created before when the raw dataset was processed, are used during training in order to create trained neural networks that are then stored. The training process involves 2000 epochs (number of times a network passes backwards and forwards) and is repeated 10 times, resulting in 10 trained neural networks with different loss functions. When training is completed, the cumulative density function is calculated along with the log-logistic probability distribution of squared errors 4.5. These functions will be very important when comparing the original measurement with the calculated prediction, as it will determine the probability of the measurement is an anomaly. If a model is not trained correctly, it means that anomaly detection will not perform as accurately and can wildly influence forecast calculation.

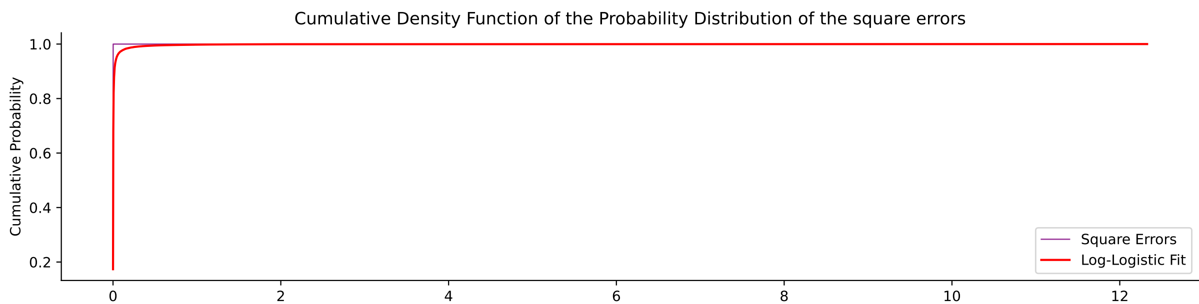

Figure 4.4: CDF of the probability distribution of the square errors

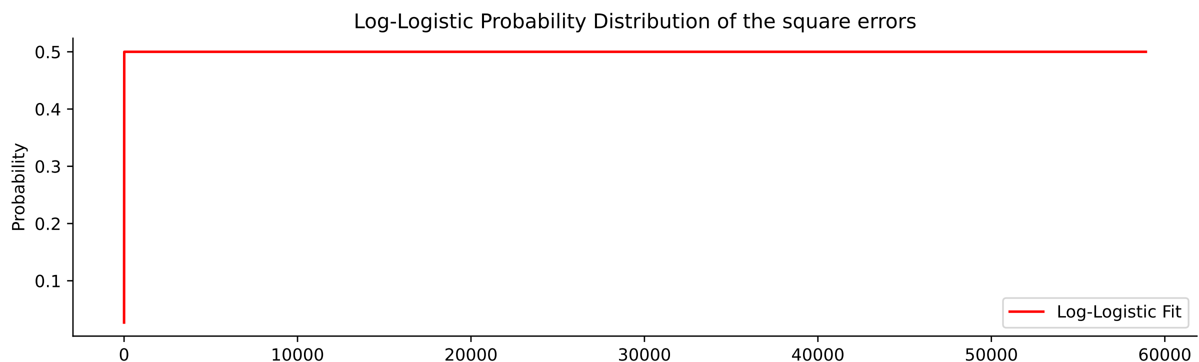

Figure 4.5: Log-logistic probability distribution of the square errors

4.3 Framework overview

This section will explain how the framework was divided into modules to be more easily understandable for future framework iterations and important files, such as the configuration file that needs to be filled out before the framework's execution. These changes also came with some code upgrades from the previous framework, which will also be discussed herein.

We will also analyze new developments that came with this thesis, like the implementation of a full dashboard, an API and the application of Docker in this work. Each one of these three new additions came with some obstacles that we were able to overcome due to some extensive investigation about how these technologies work, especially linked with Docker.

4.3.1 Server

The server of the framework consists of multiple modules, each with their own responsibilities. It is where predictions are made and need to go through some steps before we begin to forecast values.

The first step is client-server communication. The previous version implemented a communication protocol with the Pyro4 library. This is a Python library that enables the communication between objects over the network. To simplify and generalize the communication between a client and the server, it was decided to restructure that connection, deprecating the use of the library Pyro4 by invoking the use of TCP sockets. This meant that we needed to create a Server class with methods for TCP/IP socket connections. For this, an address and port are needed, which are given by the user in the configuration file (section 3.2.2). Two files take care of these connections, *Server.py* and *sock_utils.py*.

- **Server.py** - Class to communicate with a TCP server. It implements methods to establish a connection, send a message and end the connection. This module is used by our client simulator to communicate with the framework;
- **sock_utils.py** - Module with important methods to create a TCP connection and receive messages.

As part of code reorganization, new methods and logical strategies were implemented. Starting with the initialization script *app_server.py*, where we implemented two threads, one for communication and another for processing data. Once we initialize this script, it receives the execution configuration file and according to its information, it will create a SensorHandler instance which is the main class in our framework that will take care of starting forecast calculation. Then, we start the two threads ready to receive and process data from the client. To take care of mutual exclusion problems, we implemented a new method that uses Pythons Condition Locks to deal with these problems. The decision to use Condition Locks instead of Semaphores was because conditions take up less stack space than Semaphores despite having the same function (synchronized access to shared resources). However, Conditions use a lot of processor time as the condition is being checked repeatedly until it is passed.

These threads, as mentioned before, have two different objectives. The communication threads only receive messages from a client and store them in a queue that only the processing thread has access to. Once the condition lock is notified and released, the processing thread access the queue, now with data, and sends it to the SensorHandler instance for further processing.

In the next sections, we will explain in detail the primary modules that was made and remade during the code reorganization.

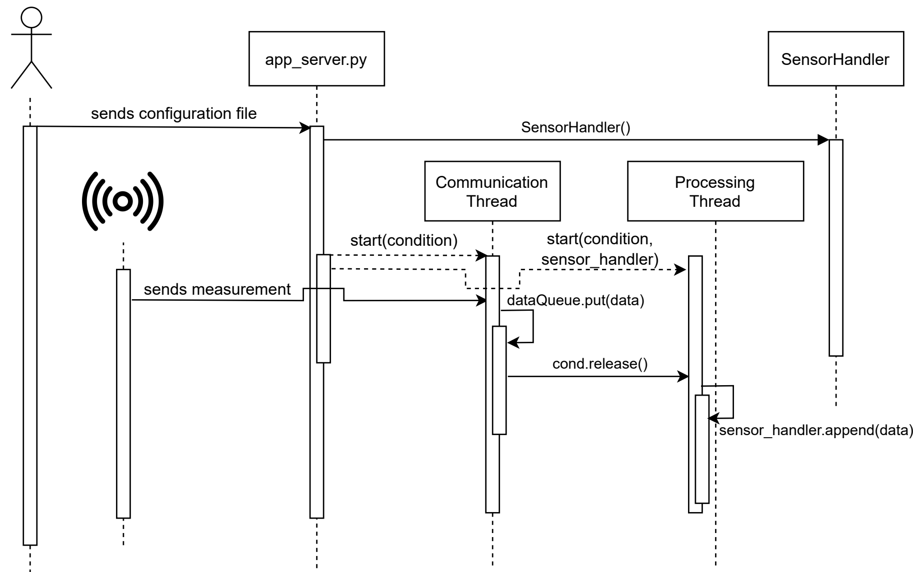

Figure 4.6: Sequence diagram of *app_server.py*

SensorHandler

The SensorHandler class is the centre of the framework. It is its core where data is redirected to other modules. It holds information from the configuration file and it handles every variable to be taken into account during processing. This class also has an object from the *PredictionBlock* and *QualityBlock* classes that will be explained in the next sections. Once a measurement is sent to the *SensorHandler* class, it is appended into a prediction queue. This queue will be accessed when it has measurements stored inside it. The received measurement is also stored in an entity called *SensorData*. When an instance of *SensorHandler* is created, it starts two threads: the prediction thread and the quality thread.

The prediction thread is triggered when there is a value in the prediction queue. With the measurement fetched, the framework starts to make predictions after passing some conditions in the *PredictionBlock* module. When a prediction is made, that value is stored in a *SensorHandler* dictionary and the quality queue is filled with sensor information.

Just like the prediction queue, the quality queue is triggered when there is a value in the quality queue. This queue has the purpose of calculating the quality of a measurement. It is important to note that the quality queue only has values inside it if the framework was able to calculate predictions. Once inside this thread, it calculates the mean squared error, detects if a measurement is faulty or not and finally, calculates its quality, a number between 0 and 1. In the end, all this information is stored in a dictionary inside *SensorHandler* and if the received measurement is an anomaly, that value is replaced by the respective prediction.

SensorData

This class works as a dictionary with previously received measurements. It stores raw and corrected data, as well as some sensor information. When a new measurement is received in this class, it is

compared with the previously received measurement to check if there were any data omissions during the communication. If there was, a new measurement is added in those periods with a senseless value (NaN) to be later replaced. This is done with the help of the *period_time* parameter in the configuration file. We also implemented a *jitter* which allows for some delays in the communication due to congestion or other events out of hand from the framework.

PredictionBlock

The prediction class is where the framework tries to calculate a prediction. This module is accessed by the *SensorHandler* class in the prediction thread. To try to make a prediction, values need to be aligned as explained before (section 3.2.1) in order to have accurate entry vectors.

Once all of this is met, the framework searches for the best trained model (lowest loss function) of each type of model (*all*, *self* and *neighbours*). Then, for each model, with the Keras API, the framework tries to make predictions with the provided entry vectors. These predictions are appended to a list that is sent back to *SensorHandler*.

QualityBlock

QualityBlock is a class that has important methods for failure detection and quality calculation. It is in this block that the CDF function, created in the training block, is used. When checking the probability of a measurement to be an error, we load that CDF function and search for the y value given the error number (the x value). y is the probability of erroneous data. If this probability is below the *cdf_threshold* in the configuration file, then the prediction is similar to the received value. Otherwise, a fault is detected. Depending on the number of sensors, this outcome could either be, in fact, an anomaly or a physical event. For this, we check the probabilities from other models resulting in the three different options explained in section 3.2.2.

To calculate the quality number, which quantifies the level of confidence in that measurement, if the measurement is an outlier, de quality is 0. If it is not an outlier, then the quality is 1 minus the average of the quadratic errors CDF (taken from [37]):

$$q = 1 - \left(\frac{\sum_{i=0}^{n} CDF_{pi}(e(m, p_i))}{n} \right) \tag{4.1}$$

4.4 Docker

In this work, we wanted to implement an API and a new dashboard. Since the dashboard has more than one software service, the containers needed to communicate with each other. In our case, we have a total of seven containers, one for the framework, two for the API and four for the dashboard. They all communicate with each other as they need data from each other.

4.4.1 Docker flow

Figure 4.7 shows how the containers communicate with each other. A client sends data to the framework container that is executing the framework in real time. Every time it receives a measurement, it stores

that data in the MySQL container. This container is also accessed by the API container (*backend*), the Grafana container and lastly, the dashboard container. There are also two containers which are not represented in the figure as they serve as support to the API and MySQL container. These containers will be explained in the following section.

All these connections function in real time and have a quick response time which is very important in this project. Each container has its own configuration and ports available. Containers communicate with each other by designated ports. In the next section, we explain each container in detail and its configurations.

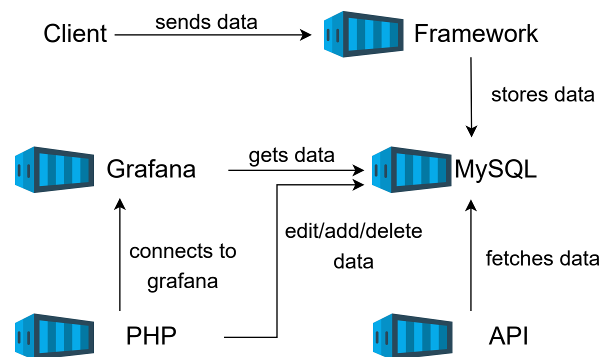

Figure 4.7: Docker flow

4.4.2 Containers and Images

To create images and containers, we need a Dockerfile. This file contained instructions as to how a Docker image is built. Each container has its own Dockerfile with different instructions, depending on the application. When this file is executed, it creates a Docker image which is then used as a template to create a Docker container. In this work, we used docker-compose to start and create our containers. A docker-compose file runs Docker containers based on the written configuration in the file.

The docker-compose file is a YAML file that defines services (containers) and their configurations. Figure 4.10 shows part of this file for this project. As we can see, the parameter **`services`** is where we represent each container we want. Every container has a name, an image that can be created by a Dockerfile, environment variable (optional), ports (optional) and volumes (optional). For the containers to communicate with each other, we need to define some volumes. Otherwise, they cannot access important data. The parameter *depends_on* expresses a dependency on other containers.

This project has seven containers:

- **`nginx`** - a web application that helps the implementation of server-side applications such as this framework. Depends on the **`backend`** container as the exposed port opens the API. Exposed port 80;

- **`backend`** - container for the API. Depends on the **`mysql`** container as it makes a lot of connections with the database to fetch data;

- **`grafana`** - container for web application Grafana. It gives the ability to create dashboards, tables and all sorts of visual graphs from data stored in a database. Depends on the **`mysql`** container and

is exposed in the port 3000;

- **framework** - a container that allocates all source code of this work's framework. Also depends on the **mysql** container and is exposed in port 9999;
- **php-apache-environment** - dashboard container. Exposed in port 80 and depends on the **mysql** container;
- **mysql** - container for the MySQL database. This is the main database of this project, and it is accessed by every container. Exposed on port 3306;
- **phpmyadmin** - tool helper for MySQL database. Exposed in port 8080 and depends on the **mysql** container.

aquaIoT - docker-compose.yaml

```
1  version: '3'
2  services:
3    web:
4      container_name: nginx
5      image: nginx
6      volumes:
7        - ./nginx/nginx.conf:/etc/nginx/templates/default.conf.template
8      environment:
9        - FLASK_SERVER_ADDR=backend:9091
10       - DB_PASSWORD=password
11       - DB_USER=aquamon
12       - DB_NAME=aquamon
13       - DB_HOST=mysql
14     ports:
15       - 80:80
16     depends_on:
17       - backend
18
19   backend:
20     container_name: app
21     build: flask
22     environment:
23       - FLASK_SERVER_PORT=9091
24       - DB_PASSWORD=password
25       - HTTP_OR_HTTPS=HTTP
26     volumes:
27       - flask:/tmp/app_data
28     restart: unless-stopped
29     depends_on:
30       - mysql
31
32 (...)
33
34 volumes:
35   flask:
36   framework:
37   mysql:
38   grafana_data:
```

Figure 4.8: Docker-compose file example

Most of these containers have custom images created by Dockerfiles. Figure 4.9 shows the Grafana Dockerfile as an example of these files.

```
aquaIoT - Dockerfile

1  FROM grafana/grafana-enterprise
2
3  # Disable Login form or not
4  ENV GF_AUTH_DISABLE_LOGIN_FORM "true"
5  # Allow anonymous authentication or not
6  ENV GF_AUTH_ANONYMOUS_ENABLED "true"
7  # Role of anonymous user
8  ENV GF_AUTH_ANONYMOUS_ORG_ROLE "Admin"
9  ADD ./grafana.ini /etc/grafana/grafana.ini
10
11 ADD provisioning/dashboard/*.yml /etc/grafana/provisioning/dashboards/
12 ADD provisioning/datasource/*.yml  /etc/grafana/provisioning/datasources/
13 ADD provisioning/dashboard/*.json /var/lib/grafana/dashboards/
```

Figure 4.9: Grafana Dockerfile

4.5 Dashboard

To make a dashboard that can show panels of sensor data and the sensor's location and still let the user configure sensors, we decided to use the following technologies: Grafana, PHP and MySQL database. However, for the first iteration of this dashboard, we used Node-RED and the InfluxDB database to show all the required panels. These two platforms had their disadvantages. For instance, Node-RED does not allow freedom in their dashboard packages, limiting our approach, and InfluxDB is a time-series database that would only be useful if we had time-series data to store, which is not the case. There are a lot of other fields that need to be referred to a specific measurement that was not doable in InfluxDB.

For these reasons, the final iteration of the dashboard was implemented using PHP, Grafana and MySQL, along with the basic technologies like HTML, CSS and JavaScript. There is only one main page with all the information needed by the user. On the left-hand side is a sidebar with a list of all available stations. When a station is clicked, a new list appears with the sensors within that station and their status. This status can be active, inactive or disabled. When a sensor is selected, the respective graphs will appear below. The user can select as many sensors as he wishes to see on the dashboard. Each of these graphs has three different types of view available: all measurements view, corrected measurements view, and raw measurements view. The all measurements view shows two different lines, one representing the corrected measurements and another representing the raw measurements. These graphs are embedded with Grafana dashboards produced based on recurring queries to the database. The user can choose which he wishes to see. The graphs are refreshed every 5 seconds, allowing the user to see data being received in real time. Three graphs are always displayed on the dashboard: the quality average meter, the number of outliers and the number of omissions. These graphs show data from all sensors and are not linked to the selected station.

There is also a map that shows where the sensors are located. In this case, since we worked directly with LNEC, the map is centred on Seixal bay. If we select a marker pinned on the map, a short description will appear with the name on the sensor.

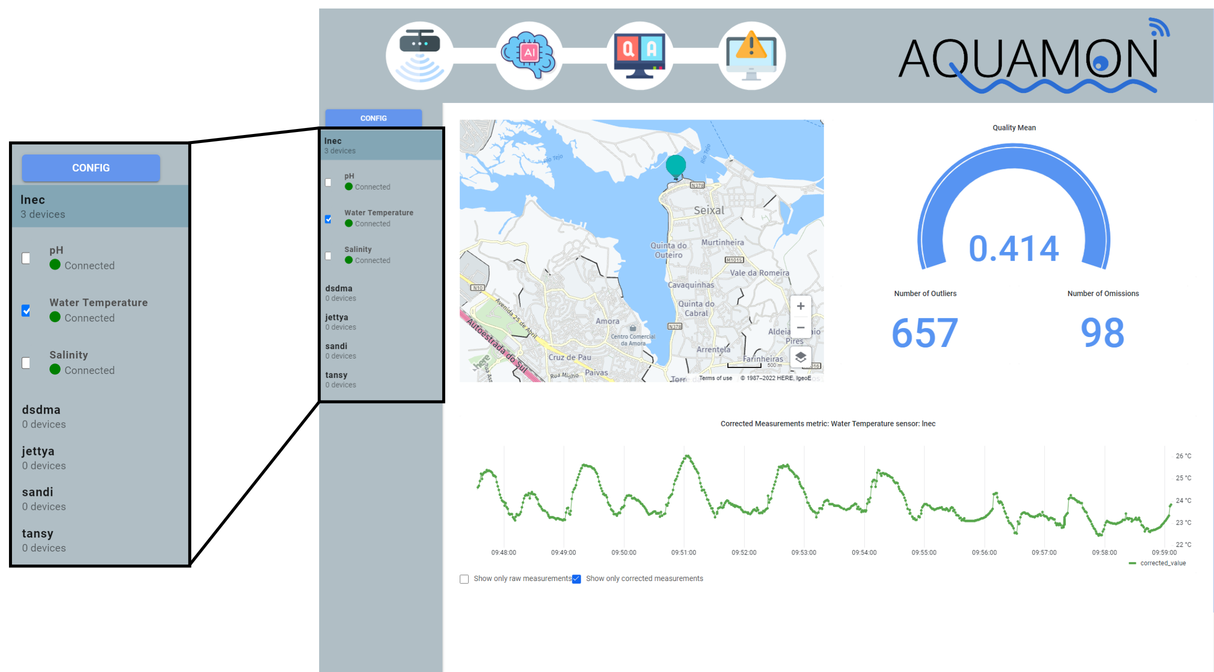

Figure 4.10: Dashboard main page

4.5.1 Sensor configuration

Just like in the framework, the dashboard allows sensor configuration. This option is located at the top of the sidebar. Here, there are 5 different options:

- **Add station** - Enables the user to create stations by providing a name, latitude, longitude and sensors available in the new station;
- **Delete station** - Enables the user to delete unwanted stations;
- **Edit station** - Allows the user to delete sensors associated with a station;
- **Add metric** - Creates a new metric. When a new metric is created, the user can create a new sensor with that metric;
- **Add device** - Adds new sensors to a station. Status and metrics are given.

All these configurations are added to the database. However, they do not alter specification in the framework, they only create, update or delete instances in the database. If the user wishes to change the framework's configuration, they will have to change the respective configuration file.

4.5.2 MySQL Database

To store incoming data we use the MySQL database. MySQL is a relational database which provides relationships between tables. We needed a database that would be able to store sensor data and their measurements. These two entities should also be related to each other, thus the decision to use a relational database. The database consists of 5 entities: sensor, metric, station, status and measurement. Figure 4.11 shows every relation between each table and their attributes. This database is accessed by almost every element in this work and is also what makes the dashboard asynchronous.

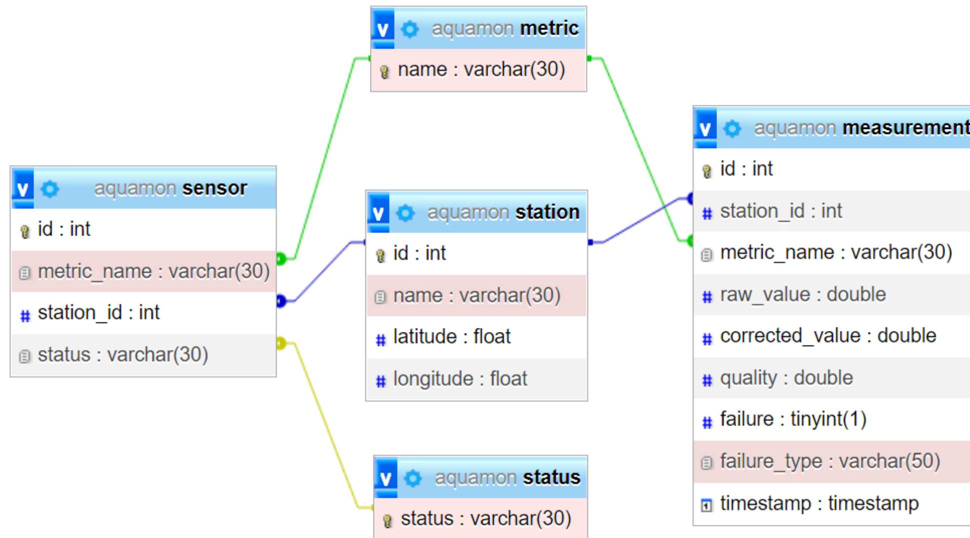

Figure 4.11: Database Relation schema

The table *Station* is used to store all the information from a station, meaning its `latitude`, `longitude` and its `name`. The primary key is, of course, the `id` that is given automatically at the moment of creation.

Table *Sensor* is the table representing a station's device, and so, it is related to table *Station*. It has attributes such as an `id` (primary key), `metric_name` (foreign key), `station_id` (foreign key) and `status` (also foreign key). This table is related to three other tables. As it uses its primary keys as columns, they turn foreign keys in this table.

Table *Metric* and *Status* only have one attribute, `name` and `status`, respectively. They must be unique and are primary keys. They both are related to table *Sensor* and table *Metric* is related to table *Measurement*.

Finally, table *Measurement* is where we store measurements directly from the framework. Here, we have specific columns that allow us to filter data easily and fast. This table is related to table *Station*, inheriting the attribute `id`, and table *Metric* with attribute `metric_name`. `raw_value` represents the received value, `corrected_value` represents the more accurate value, this can be a prediction value or a raw value, `quality` is the quality deficit of that measurement, `failure` is a boolean that tells us if that measurement is an anomaly or not, `failure_type` tells us which type of anomaly that measurement is considered and `timestamp` is the exact time that measurement was received.

Due to the amount of information a single measurement has, it turned our implementation in InfluxDB more and more difficult and inefficient as it does not allow the creation of tables. In order to have this type of depth in a single measurement, the database started to be complex because of its database structure, which did not feel adequate. It was also impossible to relate measurements from one database to the other as they are the complete opposite of each other (relational and non-relational databases).

4.5.3 Grafana

Grafana is a web application that provides graphical interactive interfaces that are connected to a database. We use this technology to present graphs in our dashboard. First, we had to configure our data source.

With the help of Docker, we were able to connect our MySQL database to Grafana. Once we have our database all setup, we are now able to create panels.

There are lots of different graphs available, from time series graphs to tables. Depending on the type of graph we want, we chose our best option. We created a total of six panels: all measurements, raw measurements, correct measurements, number of outliers, number of omissions and quality mean. All these are shown in the dashboard. For the measurement graphs, we have to create global variables in Grafana. These variables are:

- **sensor_name** - has values from the result of the query `SELECT 'name' FROM 'station';`
- **metric_name** - has values from the result of the query `SELECT 'name' FROM 'metric'`.

These variables are used in the graph's configuration. In order to have data, each graph makes a query to the database and shows the results in the graph. These variables are also very useful to create dynamic graphs, i.e, graphs that according to the user's choices in the dashboard's sidebar, show the respective graph for the correct sensor and metric.

Whenever there is an alteration in any graph's configuration, we need to save the JSON file it creates inside the provisioning directory, so that if Docker needs to reset, all changes are not lost. The same goes for the database settings.

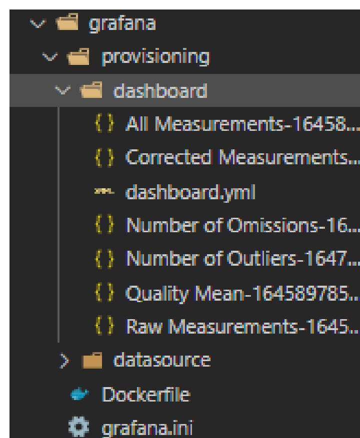

Figure 4.12: Grafana directory

4.6 API Implementation

For the API we opted to use Flask as it is a lightweight Python framework that allows quick API implementation. This API follows the REST architectural structure. Because this is a simple API that only fetches data that results from the framework, we only have one resource method, the GET method which allows the server to find data that was requested and send it back to the client. The API is constantly connecting to the MySQL database to execute queries to have data back.

As mentioned in section 3.3, `raw_measurements` is one of the options we have in the API. If we want to make an API request for the first three days since the LNEC sensor started to take water temperature measurements (01-09-2021), an API route example would be:

`http://localhost:80/api/lnec/temp/01-09-2021/3/raw_measurements`.

4.7 Framework's and Docker execution

Figure 4.13 shows how this work is organized. As we can see, there are six directories, one YAML file and three bash files. The YAML file has already been discussed before, it is where all the Docker container configuration is being done. To execute this, we have two options. We can either execute the bash files in Linux with the following command on the terminal `./setup.sh`. Or we can use Docker Desktop in either Windows or Linux systems, where we run the command `docker-compose up -d -build`. This command will start all containers in the docker-compose file and present them in the Docker Desktop software, along with their respective logs and status.

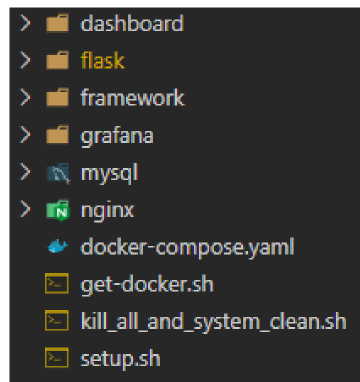

Figure 4.13: File structure

- **dashboard** - Has all the files created during the implementation of the dashboard. It is divided into four sub-directories, each for each type of file (CSS, images, JavaScript and PHP files);
- **flask** - Has the API-related files including a requirements list that is accessed when the API container is created installing every library needed and used in the Python files;
- **framework** - Here is where all framework scripts are located. This directory will be explored in depth in the following section;
- **grafana** - In here we have all the Grafana settings. If any changes need to be made, the directory *provisioning* has all the files that correspond to the database and graph configuration;
- **mysql** - Stores the SQL file with the database settings;
- **nginx** - NGINX configuration;
- **docker-compose.yaml** - Docker compose file with all container configurations;
- **get-docker.sh** - Bash file that is only used if the user does not have Docker installed;
- **kill_all_and_system_clean.sh** - Bash file that deletes all containers and images including any cache left behind. This file is useful if changes were made to a container and the user needs to reset the system;
- **setup.sh** - Bash file that starts and creates all containers according to the docker-compose file.

4.7.1 Framework directory

Going more in-depth in the framework directory, we have three more directories: **ann** directory, **data** directory and **scripts** directory. **ann** is where all trained models are located and the training checkpoints are collected. **data** is where *.csv* files are containing the datasets used during this project. The **scripts** directory is where we find all the scripts made throughout this work.

Inside the latter, we reorganized the scripts from the previous version and ended up with the following list:

- **ann_trainingsets.py** - This script is responsible for the processing of data before training. To execute this script, we need to run the following command:

  `python ./scripts/ann_trainingsets.py ./scripts/configuration_training.json`.

  Once executed, the script will start data processing, as explained in section 4.1;

- **ann.py** - This is where we train neural networks after processing data for training. The script is expecting the same JSON file as the **ann_trainingsets.py** script. Run command:

  `python ./scripts/ann.py ./scripts/configuration_training.py`;

- **app_server.py** - This is the main script that starts the framework. It expects the JSON file explained in section 3.2.2. Run command:

  `python ./scripts/app_server.py ./scripts/configuration_processing.json`;

- **app_simulator.py** - This script simulates a client that is sending measurements to the framework. Run command:

  `python ./scripts/app_simulator.py ./scripts/configuration_processing.json 0.1`

  The last argument passed on the command is the frequency at which the client will send measurements to the framework, in this case, every 0.1 seconds the client sends data to the server;

- **configuration_processing.json** - Configuration file for processing;

- **configuration_training.json** - Configuration file for training;

- **db_setup.py** - This script contains all interactions with the database needed by the framework. It inserts stations and measurements into the database. If the database suffers alterations, the user only needs to change this module;

- **entry_vectors.py** - Script where entry vectors are created;

- **functions.py** - This script contains important functions needed throughout the framework's execution;

- **PredictionBlock.py** - This script contains the PredictionBlock class which is responsible for calculating forecasts;

- **QualityBlock.py** - This script contains the QualityBlock class which is responsible for calculating a measurement's quality and failure detection;
- **SensorData.py** - Contains the SensorData class which stores every measurement received in a Python dictionary;
- **SensorHandler.py** - Contains the SensorHandler class which handles all sensor's variables;
- **Server.py** - Script that contains the Server class which is used for server-client communication;
- **sock_utils.py** - Script containing functions that help with server-client communications.

4.8 Summary

This chapter contains all things related to the implementation of this project. Here, we talk about how things were developed in detail, from framework improvements to the integration of Docker and the implementation of a dashboard and an API. We expose all configurations that were needed and some drawbacks that we faced during implementation.

Chapter 5

Results

This chapter describes the results we got from using the framework's new version. These results consist of validating the previous results from [37] and a real time validation with data collected by LNEC's sensors located in the Seixal bay. Finally, we will describe how raw values appear to the user in the dashboard and how they see the corrections happen in real time.

5.1 Data

The framework was implemented by using data from an environment monitorization network called SATURN Observation Network [9]. This system is located on the Columbia river between Washington and Oregon in the United States of America. The network has various sensors, but for the ANNODE framework, only four sensors were used. The used sensors are Jetty A, Lower Sand Island Light, Desdemona Sands Light and Tansy Point. Their locations are represented in Figure 5.1.

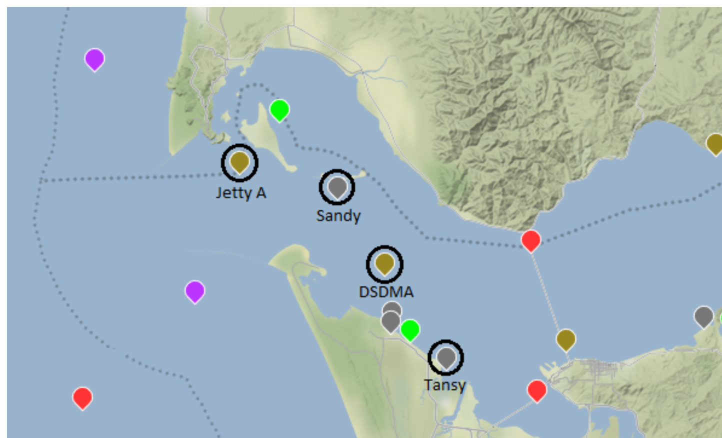

Figure 5.1: SATURN Observation Network

This dataset was used as a guideline for the results of the new version. Due to the restructuring of the code, our results needed to be the same or better than before.

On the other hand, we also used data from LNEC's sensors [40]. These sensors are located in the Seixal bay and consist of various types of variables such as water temperature, pH levels, and salinity, among others. During the implementation of this work, we only used data from one sensor at a time, different from the SATURN dataset.

This dataset suffered the same alterations as the SATURN dataset for training. LNEC gave us permission to fetch data from their API at UBEST [11].

5.2 Framework validation with previous work

To validate our improvements in the framework, we used the dataset from previous versions. This dataset is from an environmental monitoring network called SATURN Observation Network [9].

All these sensors collect water temperature data. That data was divided into three different datasets for various purposes. Each set of data was used in different stages. One was used for neural network training. Another one was used to calculate statistical distributions of errors in the neural networks, and the final set was used to test the framework. The first set had a year's worth of data as the neural networks needed to have information about each season (summer, fall, winter and spring), whereas the other datasets had information worth an entire month in different seasons of the year.

With the observation of the datasets, it was possible to conclude that the entry vectors have to consist of at least twelve hours of data to represent a full tide. For training, the dataset should consist of data from a whole year to represent the four different seasons as water temperature differs in the summer and winter, as data suffers great alterations as seasons go by. Data also differs with each tide. The more data we use to train models, the more representative data will entry vectors have, making predictions more accurate.

We trained new models using the same data as [37]. Although data was the same, we trained models using *.csv* files instead of the original *.mat* files. Training one model for one sensor would take up to three days with hardware of 16GB of RAM and an AMD Ryzen 5 3500X CPU. For each of the four sensors, three datasets were used:

- **Training set** - Comprised of data between 21-07-2009 and 05-06-2010;
- **Error statistical distribution set** - Comprised of data between 20-08-2010 and 10-10-2010;
- **Test set** - Comprised of data between 01-10-2013 and 01-01-2014.

These test datasets have outliers and periods of missing data in them for the framework to detect. *Desdemona* sensor has 44 outliers, *Tansy Point* has 11 outliers, *Lower Sand Island Light* has 1 outlier and *Jetty A* does not have any outlier. Every sensor has periods of missing data, although very hard to know how many, we can confirm that during testing the framework with a week's data, there was at least one period of data omission in each sensor. All were detected and corrected.

5.2.1 Outlier detection

In order to correctly compare results between this and the last version of the framework, the evaluation method will be the same, using detection ratio (DR) and false-positives ratio (FPR). DR is calculated by dividing the number of detected true positives by the total number of true positives and FPR is calculated by dividing the number of false positives by the total number of true positives.

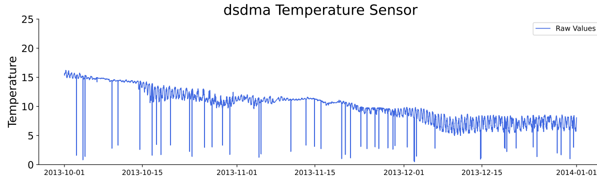

Figure 5.2: *Desdemona* raw measurements

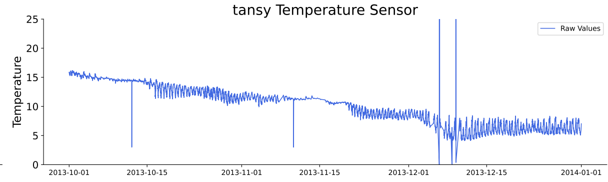

Figure 5.3: *Tansy Point* raw measurements

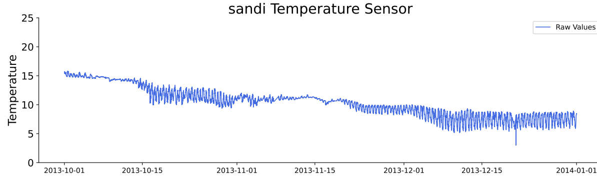

Figure 5.4: *Lower Sd* raw measurements

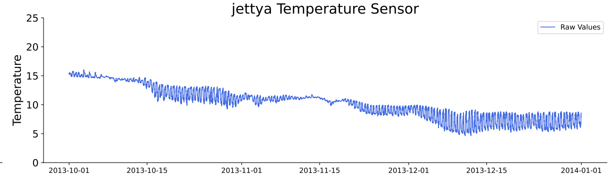

Figure 5.5: *Jetty A* raw measurements

Table 5.1: ANNODE+ results

	Jetty A			**Lower Sd**			**Desdemona**			**Tansy**		
Real outliers	**0**			**1**			**44**			**11**		
Threshold	**Detected**	**DR**	**FPR**	**Detected**	**DR**	**FPR**	**Detected**	**DR**	**FPR**	**Detected**	**DR**	**FPR**
0.9800	0	100%	0%	2	100%	0.00534%	51	100%	0.03690%	9	81.81%	0%
0.9950	0	100%	0%	2	100%	0.00534%	49	100%	0.02636%	9	81.81%	0%
0.9960	0	100%	0%	1	100%	0%	49	100%	0.02636%	9	81.81%	0%
0.9970	0	100%	0%	1	100%	0%	49	100%	0.02636%	9	81.81%	0%
0.9980	0	100%	0%	1	100%	0%	48	100%	0.02109%	9	81.81%	0%
0.9985	0	100%	0%	1	100%	0%	47	100%	0.01582%	9	81.81%	0%

Table 5.1 show the results obtained throughout six different thresholds. Each sensor has the number of real outliers, detected outliers, and DR and FPR values. For the *Jetty A*, *Lower Sd* and *Desdemona* sensors, there is a detection ratio of 100% with the false-positives ratio close to 0% in all thresholds. These ratio values tell us that the framework can detect all outliers while keeping a low number of false outliers.

Despite these positive outcomes, *Tansy* sensor had the same results throughout the six thresholds. The framework was also not able to detect all the real outliers, having an DR value of 81.81%.

With the changes reported from the *Desdemona* sensor, the best threshold to use in our framework is 0.9985, considering the priority is the lowest FPR value possible. This threshold can get an DR value of 100% in all sensors except the *Tansy* sensor.

5.2.2 Omission failure detection

To show how the framework can detect omissions and forecast a value for those periods, we selected data from 04-10-2013 data to 07-10-2013 data and we injected a period of data omission. All four sensors have missing data omissions, we will display results from the *Desdemona* sensor.

Figure 5.6 shows a portion of the *Desdemona* sensor dataset where we can witness a period of data omissions. The red dots represent outliers that were detected by the framework, the black dots represent raw data and the blue dots represent the corrected data. As seen in the figure, the framework was able to

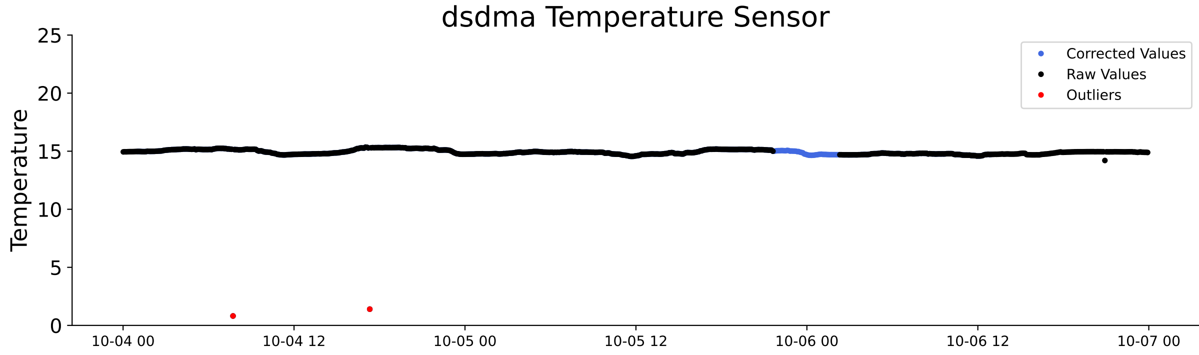

Figure 5.6: Representation of missing data periods

forecast values for the periods with no data correctly.

This behaviour continues throughout the rest of the dataset. Although it is hard to detect them, as sometimes there is only one missing measurement, this experiment showed that the framework does detect them and corrects with accurate forecasts, which was one of our main goals in this work.

5.2.3 Comparisons between versions

Table 5.2: Comparison for *Jetty A* and *Lower Sd* sensors

	Jetty A						Lower Sd					
Real outliers	0						1					
	ANNODE+			ANNODE			ANNODE+			ANNODE		
Threshold	**Detected**	**DR**	**FPR**	**Detected**	**DR**	**FPR**	**Detected**	**DR**	**FPR**	**Detected**	**DR**	**FPR**
0.9800	0	100%	0%	4	100%	0.0208%	2	100%	0.00534%	2	100%	0.00534%
0.9950	0	100%	0%	1	100%	0.0052%	2	100%	0.00534%	2	100%	0.00534%
0.9960	0	100%	0%	1	100%	0.0052%	1	100%	0%	1	100%	0%
0.9970	0	100%	0%	0	100%	0%	1	100%	0%	1	100%	0%
0.9980	0	100%	0%	0	100%	0%	1	100%	0%	1	100%	0%
0.9985	0	100%	0%	0	100%	0%	1	100%	0%	1	100%	0%

Table 5.3: Comparison for *Desdemona* and *Tansy* sensors

	Desdemona						Tansy					
Real outliers	44						11					
	ANNODE+			ANNODE			ANNODE+			ANNODE		
Threshold	**Detected**	**DR**	**FPR**	**Detected**	**DR**	**FPR**	**Detected**	**DR**	**FPR**	**Detected**	**DR**	**FPR**
0.9800	51	100%	0.03690%	52	100%	0.0422%	9	81.81%	0%	9	81.81%	0%
0.9950	49	100%	0.02636%	48	100%	0.0211%	9	81.81%	0%	9	81.81%	0%
0.9960	49	100%	0.02636%	47	100%	0.0158%	9	81.81%	0%	9	81.81%	0%
0.9970	49	100%	0.02636%	47	100%	0.0158%	9	81.81%	0%	9	81.81%	0%
0.9980	48	100%	0.02109%	45	100%	0.0053%	9	81.81%	0%	9	81.81%	0%
0.9985	47	100%	0.01582%	45	100%	0.0053%	9	81.81%	0%	9	81.81%	0%

Tables 5.2 and 5.3 show the comparison between the current version (ANNODE+) and the previous version denominated as ANNODE [37]. At first glance, we can observe a few differences in the *Desdemona* and *Jetty A* sensor. Our version detects more outliers in this sensor, resulting in a higher FPR value than the previous version (0.0053%). This tells us that although our framework is capable of detecting real outliers, it can sometimes detect false outliers. A possible reason for this behaviour could be the replacement of missing values. It was noted that the previous version did not replace missing values, thus the

respective entry vectors could be filled with even older measurements. In this case, we replace missing data with calculated forecasts which sometimes can be inaccurate. Nevertheless, entry vectors are filled with these values resulting in a different outlier detection than the previous version. *Jetty A*, *Lower Sd* and *Tansy* sensors have the same results as before.

Comparing the results from the *Jetty A* sensor, there is a difference when using the 0.98 threshold. The previous version detected 3 outliers whereas our version did not detect any outliers. This shows that there was an improvement when developing the current version. This could also mean that the models were better trained to result in better failure detection.

In our implementation, the best threshold to detect anomalies is 0.9985. The previous version showed reliability at a threshold of 0.998, however, in the new version we had a slight increase in the FPR ratio in that threshold, making it not as accurate as before. Even with this increase in the ratio, we can see improvements in our version in the *Jetty A* sensor, in which our framework was more accurate. With these results, we can say that our implementation has a slightly better performance compared with previous versions of this work.

5.3 Validation with LNEC's data

5.3.1 Outlier and omission detection

To validate our framework with LNEC's data, we fetched 3 months of data from their API. For training, we used data from the first of July to the last day of August. Then, to test the framework, we used data from the first week of September. To check if the framework would be able to detect anomalies, we injected multiple anomalies into the dataset, 3 outliers and one period of missing data. To emulate sensors, we implemented a simulator which would send each measurement to our server in order to process them. In this case, we only have one sensor sending data to the framework.

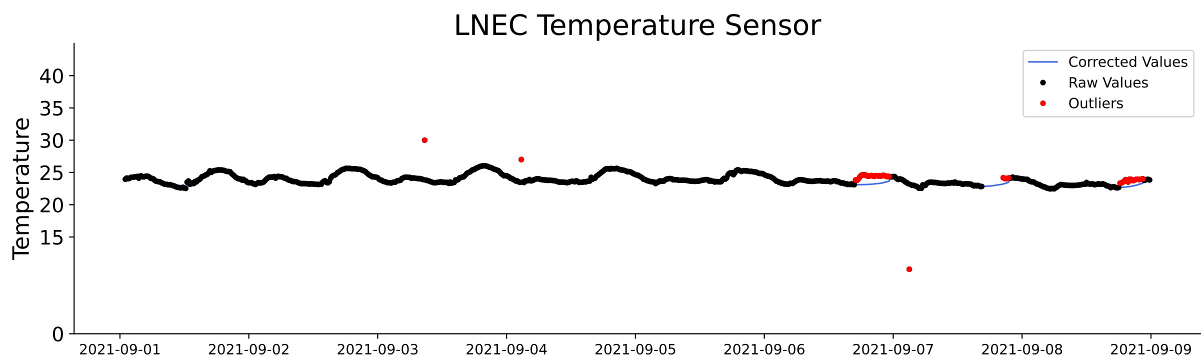

Figure 5.7: Framework's outcome with data from the Seixal bay

We observed very promising results, with the framework replacing faulty values with their respective forecasts. Since we used the previous two months' worth of data to train models, the calculation of forecasts was very accurate. The framework was also able to detect missing data in datasets as well as forecast values during that time thanks to the models that were trained beforehand.

Figure 5.7 shows the results we were able to get with a dataset of a whole week of raw measurements. As we can see, the framework correctly detected all 3 outliers and corrected those values with their

respective forecast. The same happened in periods where there was no data (missing data). The period of data omission was detected and corrected with forecasts. However, the framework detected many false positive outliers. Although the corrected measurements are plausible, they are not 100% correct. This happened due to the small input of data for model training.

To prove that this framework is capable of learning and forecast different types of variables, we collected data from 17-06-2021 to 17-08-2021 of salinity levels from one of LNEC's sensors. With this dataset we trained new models and then with data from 18-08-2021 till 18-10-2021 we tested the framework. This test dataset had one outlier to be detected.

Table 5.4: Results from LNEC's sensor with two types of variables

	LNEC's sensor					
Real outliers	**3**			**1**		
Type of variable	**Water Temperature**			**Salinity**		
Threshold	**Detected**	**DR**	**FPR**	**Detected**	**DR**	**FPR**
0.9800	3	100%	0%	1	100%	0%
0.9950	3	100%	0%	1	100%	0%
0.9960	3	100%	0%	1	100%	0%
0.9970	3	100%	0%	1	100%	0%
0.9980	3	100%	0%	1	100%	0%
0.9985	3	100%	0%	1	100%	0%

These results only contemplate datasets with outliers and no missing data. As we can see from Table 5.4, all outliers are detected with no false positive outliers throughout the six different thresholds. It is important to note that the results shown in Figure 5.7 show different outcomes because there were parts of data omission, which has direct influence in the forecasts.

5.3.2 Different approaches for entry vector creation

We mentioned the idea of different approaches to entry vector creation. We added two new approaches to the framework: the linear approach and the last-ten approach. We did not spot significant differences in failure detection or forecast calculation when we ran these different approaches. Figure 5.8 shows the calculated forecasts from each approach, where the black line is the forecasts from the exponential approach, and the blue line corresponds to the linear approach forecasts. The red line corresponds to the last ten approach. As we can see, the three lines overlap, meaning all approaches calculate the exact estimates. When these lines are separated, the main difference in these approaches is the first few forecasts, whereas the exponential approach is the only approach able to forecast the first values.

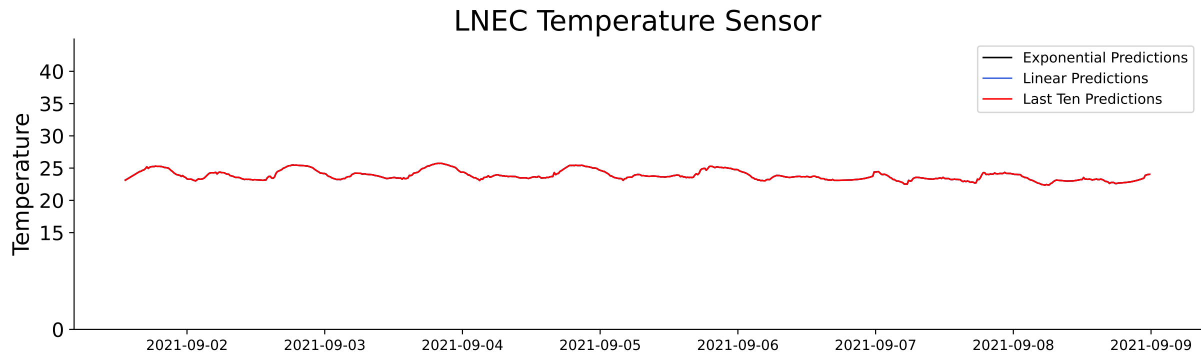

Figure 5.8: Forecast calculation with all approaches

5.4 Dashboard data and real time validation

When simulating a real time occasion, we can observe the framework's behaviour through the dashboard and its graphs. As mentioned before, Grafana is responsible for the implementation of dashboards and thus showing data in real time. These graphs are automatically refreshed allowing the user to see measurements being corrected.

Figure 5.9 shows the All Measurements panel in which the variables *sensor_name* and *metric_name* have the value *lnec* and *Water Temperature* respectively. This means that the graph shows *Water Temperature* data from the sensor called *lnec*. The All Measurements graph allows the user to see both raw measurements and corrected measurements. In the figure, we can detect all the injected anomalies, the same as in figure 5.7. This is the graph that appears in the dashboard when the user selects these options on the sidebar. The yellow line represents the corrected measurements whereas the green line represents the raw measurements. Of course, this graph is constantly being updated with new received measurements and giving the user a more visual approach to what the framework does in real time.

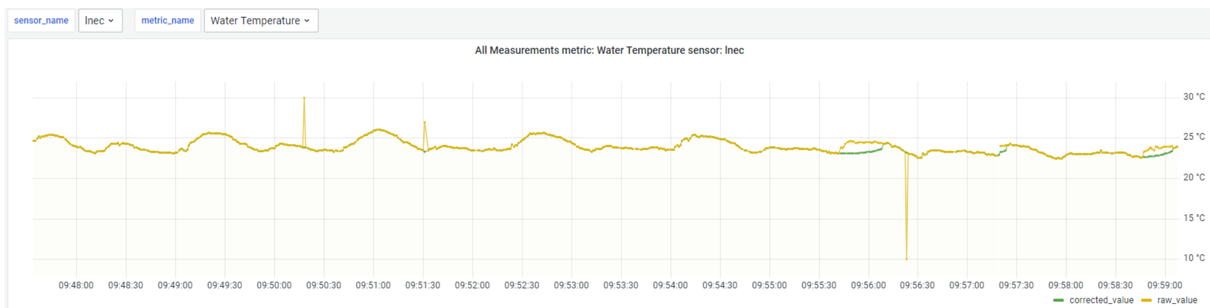

Figure 5.9: All Measurements panel in Grafana

5.5 API results

As detailed in section 3.3, we have a total of six routes available. To show the results of some routes, we used the tool Postman [7] which makes requests to APIs and shows their response. As we used LNEC's dataset to test the framework, those measurements are the ones stores in the database, so we will use those dates to fetch results. For this we need to make sure that all Docker containers are up and running.

/api/lnec/temp/01-09-2021/1/raw_measurements


```
{
  "results": [
    {
      "value": 24.33,
      "time": "2021-09-01 13:15:00",
      "sensor": "lnec",
      "metric": "Water Temperature"
    },
    {
      "value": 24.41,
      "time": "2021-09-01 13:30:00",
      "sensor": "lnec",
      "metric": "Water Temperature"
    },
    {
      "value": 24.48,
      "time": "2021-09-01 13:45:00",
      "sensor": "lnec",
      "metric": "Water Temperature"
    }
    (...)

]}
```


/api/lnec/temp/01-09-2021/1/quality


```
{
  "quality_mean": 0.451838761039042
}
```


/api/lnec/temp/05-09-2021/2/anomalies


```
{
  "anomalies": [
    {
      "value": 23.8,
      "time": "2021-09-05 07:30:00",
      "sensor": "lnec",
      "metric": "Water Temperature",
      "anomaly_type": "outlier"
    },
```

```
    {
      "value": 23.66,
      "time": "2021-09-05 08:30:00",
      "sensor": "lnec",
      "metric": "Water Temperature",
      "anomaly_type": "omission"
    },
    (...)
]}
```


5.6 Summary

This chapter presented the results we were able to obtain using the latest version of the framework. In Section 5.1 we explained the two different datasets that were used during the project. Section 5.2 shows the results using the SATURN dataset and compares them with the results obtained by the previous version in [37]. Section 5.3 shows the results obtained by the framework using the LNEC's dataset. Section 5.4 shows how data is seen in the dashboard and Section 5.5 details some API requests and their respective responses.

Chapter 6

Conclusion

Going back to the goals defined for this thesis, we conclude that all were met. We successfully improved the previous version of the framework by adding omission detection and replacement for correct values (calculated forecasts). We also implemented an API, similar to LNEC's API, that followed the RESTful services architectures and provided results from the framework. Finally, a graphical interface was implemented, supporting the framework by visualizing data in graphs that are easy to read by any user.

The ANNODE+ proposed in this thesis followed the architecture proposed in sections 3.1 and 3.2. First, we started by reconfiguring the framework's structure. During this phase, we observed some difficulties in calculating accurate forecasts. This happened because the previous version was focused on one type of dataset, which was something we had to re-implement. The next step was to develop a new failure detection method, which came to be an omission detection method. This method uses a *jitter* as a variation of time the measurement could exceed.

When the framework was able to detect and correct anomalies with one sensor, we started to configure the framework to be able to receive data from multiple data sources. For this, we introduced the idea of services where each would be responsible for different parts of data processing. Thus, Communication and Processing services were created. Inside the Processing Service, another service is responsible for forecast calculation and quality assurance.

After the improvements to the framework were completed, we started creating a platform that would secure the needs of the framework and the dashboard and its features. We started implementing Docker containers for every feature needed, including a MySQL database and the Grafana software. This allowed us to connect the framework to the dashboard, which uses graphs from Grafana for data visualization. With Docker containers, we also implemented an API that fetches data the framework gives as an output.

When comparing the framework to the previous version, we can see an improvement in outlier detection. ANNODE+ can detect fewer false positive outliers, which is progress. We also showed that the framework is capable of receiving different variables and can be used in those use cases if needed. The dashboard is also a great way to see the framework's behaviour, where the user can see failure detection and correction in real-time.

We conclude that this work is an improvement to the previous version of the framework, as the results showed a greater precision in outlier detection. We also added an omission data detection method to the framework, which enriched its purpose.

6.1 Future work

Even though we managed to fulfil the project's requirements, several points should be improved to make the framework more reliable. When omissions of data occur, the calculated forecasts are not always accurate. Due to the lack of big datasets from LNEC sensors, we sometimes faced occasions where the forecasts were not on track with the rest of the dataset. A solution for this could be better model training to facilitate this process.

Another important point is the integration of the API to send measurements and configure the framework from there. For now, the user needs to configure the framework using JSON files and then deploy the changes to Docker.

Finally, since in [29] G. Jesus was able to implement the detection of other anomalies such as offsets and drifts, these methods should be added to the current version.

6.2 Publications

Throughout the development of this thesis, two papers were written and accepted in two conferences, Ada Europe and INForum.

- Artificial Neural Networks for Real-Time Data Quality Assurance [43];
- ANNODE+: Outlier and Omission detection framework based on Machine Learning [42]

Bibliography

[1] Docker hub. https://hub.docker.com/.

[2] Elasticsearch. https://www.elastic.co/elasticsearch/.

[3] Grafana. https://grafana.com/.

[4] Keras. https://keras.io/.

[5] Kibana. https://www.elastic.co/kibana/.

[6] Node-red. https://nodered.org/.

[7] Postman. https://www.postman.com/.

[8] Python remote objects - 4.81. https://pyro4.readthedocs.io/en/stable/.

[9] Saturn observation network: Endurance stations. http://www.stccmop.org/datamart/observation_network.

[10] Tensor flow. https://www.tensorflow.org/.

[11] Ubest. http://ubest.lnec.pt/.

[12] Alireza Akhgar, Davood Toghraie, Nima Sina, and Masoud Afrand. Developing dissimilar artificial neural networks (anns) to prediction the thermal conductivity of mwcnt-tio2/water-ethylene glycol hybrid nanofluid. *Powder Technology*, 355:602–610, 2019.

[13] LoRa Aliance. Lorawan, what is it? a technical overview of lora and lorawan. 2019.

[14] George Bebis and Michael Georgiopoulos. Feed-forward neural networks. *IEEE Potentials*, 13(4):27–31, 1994.

[15] Bruno Brentan, Gustavo Meirelles, Manuel Herrera, Edevar Luvizotto Jr, and Joaquín Izquierdo. Correlation analysis of water demand and predictive variables for short-term forecasting models. *Mathematical Problems in Engineering*, 2017, 12 2017.

[16] Brilliant. Feedforward neural networks. https://brilliant.org/wiki/feedforward-neural-networks/,.

[17] Li Cai and Yangyong Zhu. The challenges of data quality and data quality assessment in the big data era. *Data Science Journal*, 14, 05 2015.

[18] António Casimiro, J Cecılio, Pedro Ferreira, Anabela Oliveira, Paula Freire, Marta Rodrigues, and Luís Almeida. Aquamon–a dependable monitoring platform based on wireless sensor networks for water environments. In *Proceedings of the 38th International Conference on Computer Safety, Reliability and Security*, 2019.

[19] José Cecílio. Aquamesh: A low-power wide-area mesh network protocol for remote monitoring applications in water environments. In *IECON 2021 47th Annual Conference of the IEEE Industrial Electronics Society*, pages 1–6, 2021.

[20] Marcello Cinque, Antonio Coronato, and Alessandro Testa. Dependable services for mobile health monitoring systems. *Int. J. Ambient Comput. Intell.*, 4(1):115, jan 2012.

[21] Gonçalo João Vitorino de Jesus. *A dependability framework for WSN-based aquatic monitoring systems*. PhD thesis, Universidade de Lisboa (Portugal), 2019.

[22] Fred Douglis and Jason Nieh. Microservices and containers. *IEEE Internet Computing*, 23(6):5–6, 2019.

[23] Joao Gomes, Marta Rodrigues, Alberto Azevedo, Gonçalo Jesus, João Rogeiro, and Anabela Oliveira. Managing a coastal sensors network in a nowcast-forecast information system. In *2013 Eighth International Conference on Broadband and Wireless Computing, Communication and Applications*, pages 518–523. IEEE, 2013.

[24] Samer Hijazi, Rishi Kumar, Chris Rowen, et al. Using convolutional neural networks for image recognition. *Cadence Design Systems Inc.: San Jose, CA, USA*, 9, 2015.

[25] International Business Machines Corporation - IBM. Neural networks, 2020. `https://www.ibm.com/cloud/learn/neural-networks`,.

[26] International Business Machines Corporation - IBM. What are convolutional neural networks?, 2020. `https://www.ibm.com/cloud/learn/convolutional-neural-networks`,.

[27] International Business Machines Corporation - IBM. What are recurrent neural networks?, 2020. `https://www.ibm.com/cloud/learn/recurrent-neural-networks`,.

[28] Gonçalo Jesus, António Casimiro, and Anabela Oliveira. A survey on data quality for dependable monitoring in wireless sensor networks. *Sensors*, 17(9):2010, 2017.

[29] Gonçalo Jesus, António Casimiro, and Anabela Oliveira. Dependable outlier detection in harsh environments monitoring systems. In Barbara Gallina, Amund Skavhaug, Erwin Schoitsch, and Friedemann Bitsch, editors, *Computer Safety, Reliability, and Security*, pages 224–233, Cham, 2018. Springer International Publishing.

[30] Gonçalo Jesus, António Casimiro, and Anabela Oliveira. Using machine learning for dependable outlier detection in environmental monitoring systems. *ACM Trans. Cyber-Phys. Syst.*, 5(3), jul 2021.

[31] Leslie Lamport. *LaTeX - A Document Preparation System: User's Guide and Reference Manual, Second Edition*. Pearson / Prentice Hall, New York, 1994.

[32] Duc C Le and Nur Zincir-Heywood. A frontier: Dependable, reliable and secure machine learning for network/system management. *Journal of Network and Systems Management*, 28(4):827–849, 2020.

[33] Luthfi Ramadhan. Radial basis function neural network simplified, 2021. `https://towardsdatascience.com/radial-basis-function-neural-network-simplified-6f26e3d5e04d`,.

[34] A. Messai, A. Mellit, I. Abdellani, and A. Massi Pavan. On-line fault detection of a fuel rod temperature measurement sensor in a nuclear reactor core using anns. *Progress in Nuclear Energy*, 79:8–21, 2015.

[35] Huynh AD Nguyen, Lanh V Nguyen, and Quang P Ha. Iot-enabled dependable co-located low-cost sensing for construction site monitoring. In *37th International Symposium on Automation and Robotics in Construction*. International Association for Automation and Robotics in Construction (IAARC), 2020.

[36] B. O'Flyrm, R. Martinez, John Cleary, Conor Slater, F. Regan, Dermot Diamond, and Heather Murphy. Smartcoast: A wireless sensor network for water quality monitoring. pages 815 – 816, 11 2007.

[37] João Penim. Implementação de soluções para confiabilidade de dados em redes de sensores sem fios. 2020.

[38] Šarūnas Raudys. Evolution and generalization of a single neurone: I. single-layer perceptron as seven statistical classifiers. *Neural Networks*, 11(2):283–296, 1998.

[39] M Rodrigues, J Costa, G Jesus, AB Fortunato, J Rogeiro, J Gomes, A Oliveira, and LM David. Application of an estuarine and coastal nowcast-forecast information system to the tagus estuary. *Proceedings of the 6th SCACR, Lisbon*, 2013.

[40] Seara.com. Laboratório nacional de engenharia civil. `http://www.lnec.pt/pt/`.

[41] Seb. Understanding backpropagation with gradient descent. `https://programmathically.com/understanding-backpropagation-with-gradient-descent/`.

[42] Casimiro A. Cecílio J. Sousa, I. Annode+: Outlier and omission detection framework based on machine learning. *INForum 2022 - Atas do 13ž Simpósio de Informática*, 2022.

[43] Casimiro A. Cecílio J. Sousa, I. Artificial neural networks for real-time data quality assurance. *Ada User Journal*, 43:117–120, 2022.

[44] Hui Teh, Andreas Kempa-Liehr, and Kevin Wang. Sensor data quality: a systematic review. *Journal of Big Data*, 7, 02 2020.

[45] A. M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 01 1937.

[46] Howard Veregin. Data quality parameters. *Geographical information systems*, 1:177–189, 1999.

[47] Richard Wang, Henry Kon, and Stuart Madnick. Data quality requirements analysis and modeling. pages 670 – 677, 05 1993.

[48] Guoqiang Peter Zhang. Neural networks for classification: a survey. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 30(4):451–462, 2000.