Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Automated data validation in model-driven IoT Applications

Christian Müller

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr. Holger Schwarz
Supervisor:	Daniel Del Gaudio, M.Sc.
Commenced:	September 2, 2022
Completed:	March 2, 2023

Abstract

Building large IoT applications using a model-driven approach, has become an important methodology for building such applications. A tool intended to help with this process is the IoT Application Modelling tool developed at the University of Stuttgart. However, this tool is lacking validation capabilities. To implement validation in a (semi) automated manner, this work proposes an approach to assist the modeller with the task of selecting the best suited outlier detection method for a context model. based on existing or accumulated data. To achieve this a semi-automated wizard, integrated into the IoT Application Modelling Tool is proposed. Furthermore, concepts on how the resulting outlier detection can be deployed on the IoT infrastructure are discussed. The provided value and the usability of the tool are evaluated using a survey on a small set of researchers and IT professionals. The result of this survey have shown, that the proposed approach does partially automate and simplify the process of choosing an machine-learning based outlier detection method.

Kurzfassung

Die Entwicklung großer IoT-Anwendungen mit Hilfe eines modellgeführten Ansatzes ist zu einer wichtigen Option für die Entwicklung solcher Anwendungen geworden. Ein Tool, das bei diesem Prozess helfen soll, ist das an der Universität Stuttgart entwickelte IoT Application Modelling Tool. Jedoch fehlt es diesem Tool an Validierungsmöglichkeiten, zum Erkennen von unerwarteten Messwerten. Um die Validierung, sowie deren Konfiguration, möglichst stark zu Automatisieren, wird in dieser Arbeit ein Ansatz vorgeschlagen, der den Modellierer bei der Auswahl der am besten geeigneten Erkennungsmethode für unerwartete Messwerte, sogenannte Ausreißer, unterstützt. Dafür wird ein halbautomatischer Assistent vorgeschlagen, welcher auf basis esistierender Daten eine passende Erkennungsmethode vorschlägt. Darüber hinaus, werden Konzepte diskutiert, wie die resultierende Validierung in der IoT-Infrastruktur eingesetzt werden kann. Der vorgestellte Assistent für Vorschläge wird als Komponente des IoT Application Modelling Tools implementiert. Der Nutzen und die Benutzerfreundlichkeit des Tools werden anhand einer Umfrage unter einer kleinen Gruppe unter Forschern und IT-Fachleuten bewertet. Duch diese wurde festgestellt, dass der vorgeschlagene Ansatz den Prozess der Auswahl einer auf maschinellem Lernen basierenden Methode zur Ausreißererkennung teilweise automatisiert und vereinfacht.

Contents

List of Figures

1 Introduction

Over the past decades computing devices and the internet became more and more ubiquitous, some examples for this are Smartphones and Smartwatches. However such devices, mostly used for personal computing needs, like browsing the web or chatting with friends, represent just a small portion in regards to how ubiquitous the internet has become. This development can also be observed in may other areas, like the Smart home, where small computing devices are used to interact with devices in the home. One component of the Smart home is home automation, where special devices can be programmed to perform different actions based upon certain inputs, like time or temperature measurements. For example, in such an application one may build an automated system, that automatically opens and closes the window blinds, based on the time of day. Another example could be a system that automatically turns on the ventilation, if the air quality in a room gets bad, for example based on the percentage of carbon dioxide in the air. These home automation use-cases show another huge contribution, to the ubiquitousness of the internet and computation devices. Since, the devices described beforehand are most likely connected to the internet and can be controlled remotely. Someone could, for example, open the window blinds while being thousands of kilometres away from home.

The smart home scenarios can be generalized into other environments, like, industrial or agricultural environments, such as production lines or greenhouses. In those environments devices that produce measurements, commonly referred to as sensors, and devices that perform an action upon a request, commonly referred to as actuators are connected together via the internet. This interconnection between a lot of different small devices, such as sensors, machines and displays is generally referred to as Internet of Things (IoT), the context of this work.

In a generalized IoT environment there are sensors, producing data, mostly measurements, and actuators that perform actions based on the sensor data they receive. Since, building huge IoT applications can become very confusing, to improve this, researchers at the Institute for Parallel and Distributed Systems (IPVS) of the University of Stuttgart propose a model driven approach, where the modeller models the data flow using a modelling program. This data-flow between sensors and actuators in an IoT environment can be modelled using a modelling tool created by them. The tool called IoT Application Modelling Tool enables the modeller to model the connections between sensors and actuators using a browser-based modelling tool. However, the tool does not include a mean to check whether or not sensor measurements are normal. Investigating, how such abnormal measurements, also referred to as outliers, can be detected will be investigated in this work.

The causes of abnormal readings are manifold, but often abnormal readings occur due to two causes. First, it may be due to an unexpected event, such as measurement of very high rainfall triggered by a thunderstorm or, to look at the industrial context, the failure of a production line because a machine is defective. Secondly, such abnormal measurements also occur when a sensor is defective, does not provide measurements for some time due to connection problems, or even fails completely.

The detection of such abnormal measurements, is referred to as outlier detection. This work has the main goal, to enable the modeller, working with the IoT Application Modelling Tool to also model validation components, that validate the data generated by sensors. To make sure, the actuators only receive data, that is valid, i.e., it does not contain any abnormal measurements and invalid measurement have been filtered out. To achieve this, this work investigates the integration of a semi-automated mechanism to assist the modeller in the selection of an outlier detection method, based on a set of existing measurements, which can be used to find valid and invalid measurements. As well as, a mechanism to deploy the resulting model, extending the IoT Application Modelling Tool.

At first some fundamentals, regarding some terms within the context will be discussed in Chapter 2. After that The tool stack around the IoT Application Modelling Tool will be introduced in Chapter 3 a concept for both making the suggestions and deploying the validations will be discussed in Chapter 4. The developed concept for the suggestion of outlier detection methods has been implemented as an extension of the IoT Application Modelling Tool. The problems we encountered and the architectural decisions we had to make in regards to the implementation the suggestion mechanism will be discussed in Chapter 5. The implemented solution has undergone an evaluation, of which the results and conclusions will be discussed after the implementation chapter in Chapter 6. After that the work is concluded by looking at other publications related to this one (Chapter 7) and, lastly, the conclusion in Chapter 8.

2 Fundamentals

The goal of this work is to integrate the detection of abnormal measurements into the application stack, described in the next section (**??**), before investigating how this can be handled some terms in regards to abnormal measurements have to be introduced. This is done in this chapter.

2.1 Outlier

The term outlier is a statistic term, that is used to name values in a dataset, of which the distance to other, non-outlier values is higher than normal. As briefly mentioned in the introduction, outliers values can have several causes, such as, (1) measurement errors, caused by a failed or defective sensor, (2) processing errors, caused by programs, that have been used to modify the data beforehand, (3) human errors, caused by humans, misconfiguring devices or entering false input values, the latter cause is not applicable to the context of this work, as data is most likely to be transmitted using machine to machine communication, (4) or new behaviour, for example, abnormally high temperatures during winter times, caused by global warming are an example. The latter kind of outliers can categorized into a special subcategory: novelties, these are outliers that cannot be considered errors, since they occur because of new, and previously unseen behaviour [Dev23b; Eff20; NIS23].

Novelties and outliers can also be considered distinct. If they are considered to be distinct, novelties usually represent an unknown cluster of values in a relatively small value range, while outliers, do not necessarily cluster and there are hardly any clusters in small value ranges [Dev23b]. However, for the context of this work, novelties and outliers are considered to be unexpected values and will therefore just be referred to as outliers.

This generic definition, in principle, can also be applied to a data stream based environment, which is encountered in the context of this work. Contrary to the assumption above, the dataset containing all accumulated measurements is constantly growing. The reason for this is the sensors that regularly provide sensor data.

A value, that cannot be classified as an outlier, in any way, is referred to as an inlier.

2.2 Outlier Detection

The outlier detection deals with the detection of outliers as described above, so that these values can be dealt with accordingly. It thereby does not matter if the data comes from a dataset, i.e., a fixed number of measurements, or a data stream, where the dataset constantly increases, as outlier detection is done on a row by row, and therefore value by value, basis.

A very simple option to detect outliers is a filter, that flags every value, that is not within a certain, predefined threshold as an outlier. This approach comes with several problems that make it non-suitable, as a generalized solution. The most obvious one, is that such a filter may only work for one specific scenario, in one environment. Manual adjustment is therefore mandatory, every time the scenario or environment changes.

2.2.1 Machine-learning based outlier detection

One common and generic solution to outlier detection, is the use of machine-learning, in order to detect outliers within a dataset. Such a outlier detection method always needs some sample data in order to adapt to the data it is expected to receive, during this adaptation, commonly referred to as training, the algorithm understands the data received and builds an internal model that is later used to perform predictions on any given dataset, whether or not a row is an outlier.

Such outlier detection methods can be grouped into three groups, that differentiate themselves by the type of data they need

- **Supervised** outlier detection methods need fully labelled data in order to be trained, this means that every row in the dataset used for training must also contain the expected prediction, i.e., whether or not the given row is expected to be an outlier or not,
- **Semi-supervised** outlier detection methods also need labelled data, however the data may only be partially labelled, meaning that only the non-outliers are labelled, and
- **Unsupervised** outlier detection methods that do not need labelled data in order to identify outliers, however they still have to be trained in order to make predictions [Eff20].

3 Introduction to the Tool stack

Since, this work has the primary goal of expanding an existing tool stack, before taking a look at how the tools can be improved, a brief introduction of the individual applications in the tool stack will be given.

The tool stack is based around a lifecycle method for decentralized IoT environments, proposed by Del Gaudio et al. [DH20] the lifecycle described in their work is based around five stages. (1) Data-flow modelling, where the flow of data between operators, like sensors and actuators is modelled, (2) Network Topology creation or modification, where the modeller selects or modifies the devices on which the sensors and actuators are hosted, (3) Data-flow execution, representing the main stage of the lifecycle, where the modelled data-flow is executed on the selected devices, (4) Device redistribution, the stage that is entered, in case the devices involved in the workflow change, and (5) Data-flow retirement, the final stage of the lifecycle that is entered in case the data-flow should be retired, for example, if it should get deleted if it is no longer needed [DH20]. The flow between the lifecycle stages is illustrated in Figure 3.1.

The tools introduced in the following sections are all involved in parts of this lifecycle.

3.1 MBP - Multi-purpose Binding and Provisioning Platform

The MBP is an open source tool[1] with the purpose to ease the deployment, monitoring and management of IoT-Devices. It provides a centralized option to orchestrate IoT Devices. It introduces terminologies, common to the IoT environment into the space of the application. The MBP allows one to create a `Device` object that represents a physical device, on such a device a program, referred to as an `Operator`, can be deployed using the MBP. For that the application connects to the device using protocols such as SSH. These operators can either act as a source of data, a sensor or a sink of data, an actuator [FHS+20].

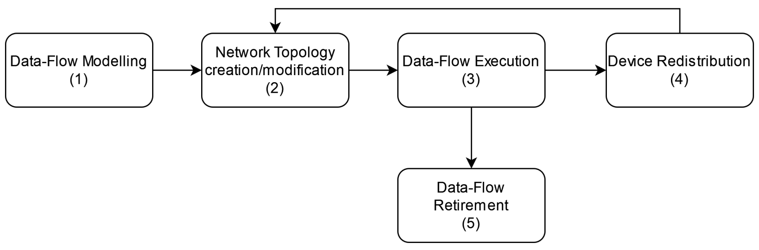


Figure 3.1: Illustration of the IoT Lifecycle proposed by Del Gaudio et al. [DH20]

[1] https://github.com/IPVS-AS/MBP

The MBP itself cannot be associated with one specific stage of the lifecycle, since it is used as a repository and gateway throughout the whole lifecycle. As it is used to interact with the devices on which the instantiated workflow is executed on, involving it in all lifecycle stages.

3.2 ME - Messaging Engine

The Messaging Engine, ME for short, is the core component of the third lifecycle stage, data-flow execution. It is a lightweight communication component that enables the transmission of data between different nodes in the underlying components of the previously modelled data-flow. It is responsible for collecting and transmitting the sensor data according to the data-flow, in order to transmit data between instances of the Messaging Engine, the Constrained Application Protocol (CoAP) [SHB14], a UDP based protocol that is similar, but more lightweight than HTTP, is used. From a technical perspective, the Messaging Engine is implemented in Python [DH20].

There are two implementations of the messaging engine, the original messaging engine proposed by Del Gaudio et al. and the, more modern, reimplementation, also called Messaging Engine 2.

3.3 IoT Application Modelling Tool

The IoT Application Modelling Tool, IAMT for short, is a modelling program that can be used to model the data-flow between operators sourced from the MBP. This modelling procedure is part of the first lifecycle stage described above. The modelling of the data-flow is done using a web-based user interface, where the modeller can visually create a directed graph, representing the intended data-flow between sensors and actuators. The sensors and actuators that can be used for modelling within the modelling tool are fetched from the MBP. This modelling represents the first lifecycle stage.
The tool also acts as a user interface for the second lifecycle stage, because the tool guides the modeller to the instantiation stage. Where the devices on which the operators, i.e., sensors and actuators, should be deployed can be selected by the modeller. From there the model can be instantiated, using the ME-Configuator.
The Modelling tool also supports monitoring of the system while it is in the execution lifecycle stage, however, since this is not relevant in the context of this work discussing these features is unnecessary [DABS22; DRH20].

3.4 ME Configurator

The responsibility of the ME Configurator, is to orchestrate the instantiation of the processing flow that has been modelled using the IoT Application Modelling tool, by connecting to the Nodes that have been selected in the modelling tool using the SSH protocols. To perform these actions, the tool first receives a manifest, from the modelling tool. This manifest contains the modelled processing flow, the devices selected for every node and other configuration parameters such as the location and the credentials for the MBP, since the MBP itself is used for managing devices.

Upon receiving the manifest, the configurator ensures that the operators selected for a node in combination with an instance of the Messaging Engine, are deployed on every device involved in the modelled processing flow [DRC20]. The tools responsibility within the lifecycle is the transition between lifecycle stage two, network topology creation or modification, and stage three, data-flow execution.

In simple terms, the ME Configurator translates the model created in the IoT Application Modelling Tool into a real world environment.

4 Developing a concept for the integration of automated outlier detection selection

As briefly discussed in the introduction, this work has several goals, including (1) the development of a (semi) automated mechanism that will suggest an outlier detection method based on existing data-measurements and the context modelled using the IoT Application Modelling Tool, (2) the pre-selection of outlier detection methods that will be suggested in the previously described mechanism and (3) the development of a concept to completely implement the whole flow, from modelling the context, over the suggestion of outlier detection methods and the deployment of the model afterwards using the ME-Configurator and the Messaging Engine.

This chapter will describe some concepts that can be used for implementing later.

4.1 What kinds of data are expected?

The type of data the concept presented here generally relies on the measurements of one or more sensors (measurement dimensions) at a certain timestamp. Over time, this results in a time series of measurements. However ,timestamps are completely ignored for the analysis in most cases, of course in some scenarios the usage of the time, or at least parts of it, for example the current hour, e.g. 14:00, may be mandatory, but these scenarios are special modelling scenarios in which the time can just be considered as another sensor input. The only important aspect is keeping the order of the measurements.

For simplicity it is assumed, that all measurement values are numeric and can be stored as floating point numbers.

4.2 Why Machine Learning based outlier detection?

Conventional outlier detection, such as the detection using predefined rules, can be used to detect outliers. However, methodologies like these have one huge drawback: The rules are only applicable to the current context. Reusing it is only possible in some cases, for example, the rules intended to detect an outlier in outdoor temperature measurements is only applicable in specific geographical area, e.g., temperatures of 45 °C can be considered outliers in Germany, since the highest measured temperature as of early 2023 was 41.2 °C[1] while this temperature may be normal during the summer in other geographical areas. Like Death Valley, where this temperature is roughly the average

[1] https://www.wetterdienst.de/Klima/Wetterrekorde/Deutschland/Temperatur/Max/

temperature during July and August[2]. This example shows, that such a rule based outlier detector may work in some cases, but may also require rule adjustments in others, doing these rule adjustments automatically is very complex, brings many difficulties and in the worst case is not even possible.

To resolve this problem, outlier detection methods based on machine learning exist. These methods take existing sensor measurements, and use them to train a model, that can be used to find out what is an outlier and what is not. Making the adaptation to a new environment relatively simple, the only requirement is a reasonably large set of measurements and can then retrain for the new environment. Greatly reducing the coupling between the context of the IoT application and the outlier detection method.

According to Bourkerche et al. [BZA20], machine learning based outlier detection methods can be used in a wide variety of applications and use-cases, including the detection of failures in a sensor, the detection of failures in an industrial production environment. Those types of applications show that this type of outlier detection is suitable for the IoT context, important for this work.

However, not every machine learning method performs equally well in every environment, many different constraints, such as the entropy of the data and the available compute performance may require the use of another outlier detection method, when adapting to a new environment. This is exactly the point the approach that is presented in the following is trying to solve.

4.3 Finding an approach for recommendations

Machine learning based outlier detection mechanisms simplify the problem of building an appropriate outlier detector for a specific model. However, since there are many such methods, it is necessary to provide a means by which the best method for a particular model can be found. This section deals with how these methods can be made comparable in order to make the decision easier for the modeller or even to take it away altogether.

4.3.1 Finding measurements to compare machine-learning based outlier detections

Choosing the best machine-learning based outlier detection method, for a specific environment is not easy, especially when considering the huge variety of different methods, that have been developed over the recent years. To give an impression on how many different machine-learning based outlier detection methods exist: Han et al. [HHH+22a; HHH+22b] did a performance comparison on many different outlier detection methods, in this publication alone 30 different outlier detections were investigated.
Choosing the one that's best for a specific environment, is therefore no simple task. One option could be an educated guess by someone with a lot of experience in the field, to determine which of these methods may suit the model best. Educated guesses are helpful but have some problems, including the following. Such an educated guess is not 100 per cent objective, as personal preferences can still be involved in the suggestion, even if only subliminally. In addition, it is difficult to measure

[2]https://www.nps.gov/deva/learn/nature/weather-and-climate.htm

Prediction / Actual	True (Outlier)	False (Inlier)
True (Outlier)	True Outliers (TP)	False Outliers (FP)
False (Inlier)	False Inliers (FN)	True Inliers (TN)

Table 4.1: Abstract confusion matrix of an outlier detection method.

whether such a guess is actually the best one for the given environment, as there is a lack of objective metrics to measure the performance, in regards to how well-suited a given method is for the current environment.

One of most important metrics, that can be measured when working with labelled data, is the accuracy, i.e, the percentage of values that the method has predicted correctly. It is computed by calculating the number of correct measurements over all measurements. It is calculated using this formula: $Accuracy = \frac{|M_{Correct}|}{|M_{Total}|}$ where M is the set of all measurements.

With the accuracy calculated, it is possible to determine the most accurate option, that is most likely the best suited method for any given context. However, making a decision based on accuracy is difficult, since this metric is not powerful enough. A high accuracy indicates that most of the predictions made by the given method are correct. But, this metric cannot be used to determine, whether an outlier detection method tends to classify correct measurements as outliers or outliers as inliers. Depending on the scenario one of the two can be more severe than the other one, for example, this could apply, to a production process in which measurements are made on a manufactured product. If measurements deviate significantly from the norm and do not get filtered out by QA, in our case the outlier detection method, this can cause great financial or brand damage. A false positive measurement on the other hand, may only cause minimal damage, since the filtered out products may be investigated by a person whom may find out that the product marked as an outlier is actually an inlier and can be redirected to the next production step. Another option is that the product goes into recycling, resulting in loss of revenue. Depending on the value of the product this may be considered a more significant problem, then sending out a false negative, highly depending on the case that gets modelled.

Metrics that can be used to measure the aforementioned behaviour can be calculated from the number of false predictions, for both positives, sometimes also referred to as type I error and negatives, sometimes called type II error. Since, the analysis tests for outliers a positive predictions represents an outlier, where a negative prediction will represent an inlier. With these numbers a table, like the one illustrated in Table 4.1, can be filled. From this table, the more powerful, metrics can be calculated including the rates of false outliers ($\frac{FP}{FP+TN}$) and false inliers ($\frac{FN}{FN+TP}$). These metrics help to perform a case by case rating, in modelling cases such as the previously explained one [Kar22; Sur20]. From this confusion matrix further metrics can be derived, some examples are discussed next.

With these values, it is also possible to compute further metrics, like the sensitivity ($\frac{TP}{FN+TP}$), representing the rate of which a true outlier is detected as such, illustrating how likely the method is to find an outlier, and to also mark it as a true one and the specificity ($\frac{TN}{TN+FP}$), represents the rate of which the outlier detection will detect inliers as such [Gle23].

In addition, collecting the number of true negatives and true positives makes it possible, to compute and display the F_1-score. Another important metric that can be used to determine, how suitable an outlier detection method is for the specific model. This metric is calculated using this formula: $F_1 = \frac{2*TP}{2*TP+FP+FN}$, the F_1-Score is also referred to as Harmonic mean and it is an alternative accuracy measurement. It is calculated by taking the average from both the sensitivity, sometimes also referred to as recall, and the precision which is calculated using this formula $\frac{TP}{FP+TP}$. It represents the ratio between true positive and false negative predictions [Bae22; Kor21; Woo22].

The accuracy can also be calculated based on the table illustrated in Table 4.1 with this formula: $\frac{TP+TN}{TP+FP+FN+TP}$.

The only main drawback is the need for labelled data, to compute the accuracy and the binary classification metrics. It does not matter, whether the user only wants to compute the accuracy on an unsupervised machine-learning based outlier detection method.

When performing an analysis for accuracy, or any derived metric from it, it is also important to make sure the outlier detection method is executed on some data it has not seen during training, to achieve this, the input dataset should be split into two smaller datasets, for example, in the ratio of $3:1$ so that $\frac{3}{4}$ of the dataset are used for training and $\frac{1}{4}$ is used for testing.

Another important metric, that can be used to compare the methods, is the time needed to perform a prediction, which is important in an internet of things environment, since computing resources are generally rather limited on IoT devices. While such performance comparisons are objective, they are highly influenced by the hardware on which the performance is evaluated. For example, if an outlier detection method uses machine-learning libraries such as TensorFlow[3]. It must be ensured that the accelerations for the graphics card or the machine-learning coprocessor, like graphics cards supporting Nvidia CUDA[4] or computers with support for Apple Metal[5], is either (1) supported by the machines that will perform the analysis in the production environment, for example, the device performing the compute intensive tasks may have a GPU, or (2) that the program performing the measurements does not use these accelerators. Metrics to compare the performance of outlier detection methods may include: (1) the time needed to train the model, (2) the total time the process of training and computing the accuracy afterwards has taken, (3) the time needed to perform a prediction on a set of rows or (4) the time needed to perform a single prediction, i.e., one row. The resulting time measurements indicate, if an outlier detection method is compute intensive, since methods that take longer to make a prediction, block the processor for longer. However, this assumption is only true if none of the outlier-detection methods use any of the previously mentioned accelerators. The prediction time metric is also important, when the modelled system is time critical and a prediction must be made within a very short time, as it can then be identified which of the methods meet this criterion.

[3]https://www.tensorflow.org/

[4]https://www.tensorflow.org/guide/gpu

[5]https://developer.apple.com/metal/tensorflow-plugin/

4.3.2 Visualizing the outlier detection method performance

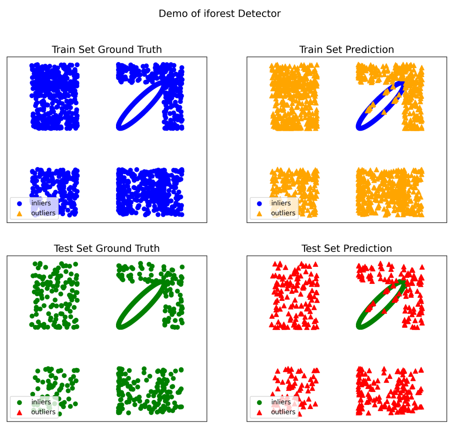
The main problem at this point are unlabelled datasets, as the metric of prediction time alone is not very useful. If any outlier method is trained with a dataset, that is not labelled, the only way to find out whether or not the method is well suited, in regards to its overall performance. Is the process of manually going through the results, which is a very tedious, monotonous and time consuming procedure. One option to improve this process is the visualization of the results. Because, this will make the differentiation between well suited methods and bad ones much easier, in most cases. However, the process cannot be completely automated, since a machine cannot visually compare the produced visualization, to identify the best suited outlier detection method. Also, decisions made based on visual impressions are not absolutely accurate. Because, it may be mandatory, to simplify the original output of the methods, for example, by binning values together, potentially resulting in some inaccuracies. This makes comparing two well suited approaches very hard, but it makes it possible to identify methods that do not perform good on the given dataset, it also allows the comparison between two different outlier detection methods. implementing such simplification measures is important, in order to ensure the data is viewable and outliers can be identified in the visualization.

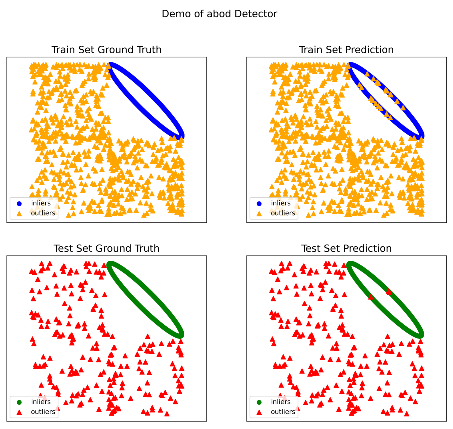
However, visualization of the measurements also has some limitations, as it is only possible to visualize up to three dimensions, due to physical constraints. But how could the outliers get visualized? To get some ideas, looking at the problem from a mathematical perspective is helpful, meaning how functions are visualized in mathematics. Since the prediction of an outlier detection method can be considered a mathematical function, that maps n real number inputs, for example, as an n-tuple, where every value of the n ones is a sensor measurement from a sensor, taken within a time frame to a binary value, that declares the given tuple, i.e., this combination of measurements is an outlier, in Equation (4.1) those functions are illustrated for one to three dimensions.

$$\begin{aligned} IsOutlier_{1D}(a) &: a \in \mathbb{R} \longmapsto \{0,1\} \\ IsOutlier_{2D}(a,b) &: (a,b) \in \mathbb{R}^2 \longmapsto \{0,1\} \\ IsOutlier_{3D}(a,b,c) &: (a,b,c) \in \mathbb{R}^3 \longmapsto \{0,1\} \end{aligned} \tag{4.1}$$

4.3.2.1 Visualization of two dimensional results

In mathematics, functions with two inputs, resulting in one output, i.e. $f : (x, y) \mapsto z$ can be visualized using a three dimensional coordinate system. However, the value range of our output value is not infinite, but limited to two values. Therefore, the number of mandatory dimensions can be be reduced to two dimensions, because the third dimension (the mapped value), being finite, can be displayed using different colours or different shapes, for measurement points. Figure 4.1[6]illustrates this kind of visualization, for an environment with two input dimensions. The illustrated visualization of such a method can be grouped into four sub graphs, two illustrating the truth values, for both the training and testing parts of the dataset, if the initial dataset was labelled, otherwise this will just show all values within the dataset. The other two two plots show which values the method declares



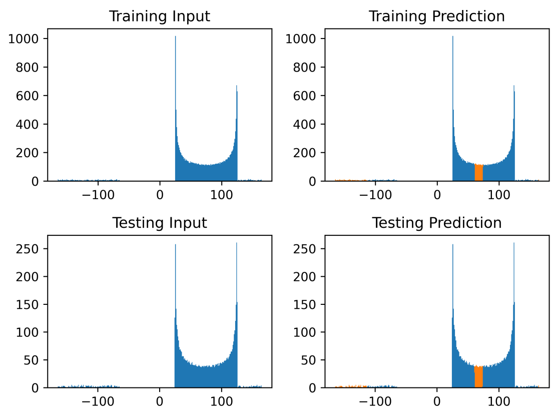


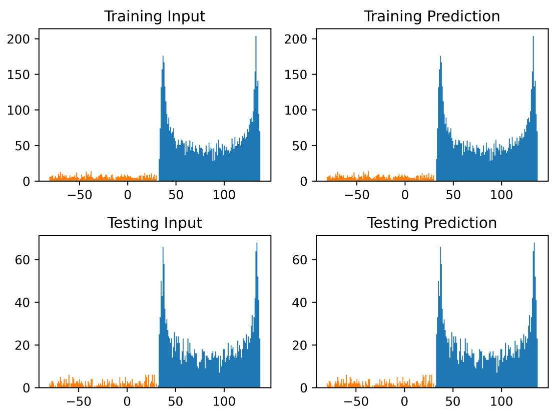
(a) Unlabelled **(b)** Labelled

Figure 4.1: Illustration of the 2-dimensional visualization for both unlabelled and labelled data

as inliers and outliers. The visualization shown in Figure 4.1 is created by a Python library called pyOD[7], that provides a huge set of different implementations of machine learning based outlier detection methods [Zha23; ZNL19].

4.3.2.2 Visualization of one dimensional results





(a) Unlabelled **(b)** Labelled

Figure 4.2: Illustration of the 1-dimensional visualization for both unlabelled and labelled data

in the one dimensional space the visualization approach, used for two inputs produces an almost understandable result, as there is only one coordinate dimension instead of two. To illustrate the values of one dimension must therefore always be one value, for example, zero. The simplest option

[7]`https://pyod.readthedocs.io/en/latest/index.html`

[7]The ellipses in the two-dimensional examples (Figure 4.1) are Lissajous Curves [Wei22] as the data is generated by two different sine waves for the first and the second sensor input, as described in Section 5.5.

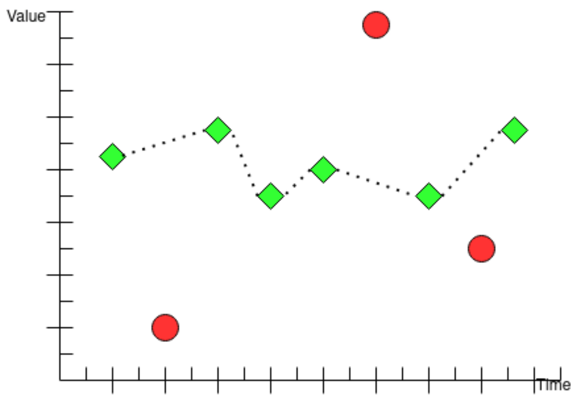


Figure 4.3: Conceptual illustration of a one dimensional time series outlier visualization

to visualize one-dimensional values, is to use a number line that runs from the minimum value of all measured values to the maximum value of all measured values. This is basically identical to the option of using the two dimensional visualization, for a one dimensional space. However, values could be illustrated better, by increasing the height of the number line, making it easier to identify outliers. On this number line, outlier and inlier are then displayed with different colours or dot shapes. However, displaying data in this manner comes with some problems, for example, the spread of data along the range between minimum and maximum is rather difficult as most inliers will, most likely be grouped within one or multiple small ranges within the large range of the global minimum and maximum of the dataset, to also contain the outliers. making this kind of vizalitzaion hard to display, especially on relatively small sized display. Therefore, other options may be considered.

One addition to the number line is the histogram, in which the values can be binned into a smaller number that is easier to render. For example, binning the data range into a 100 buckets. However this reduces the accuracy of the image, which in turn could make decision-making harder. Another problem with histograms is the fact that outliers may become very hard to see if the dataset contains many rows, of whom the measurement values are in a small range, e.g. room temperature measurements. However, this problem can be resolved by using a logarithmic scale. An example for a histogram, with a linear scale, is illustrated in Figure 4.2.

Another option could be a time-series diagram, since we assume that the data provided are measurements in the correct time-series order. Assuming the dataset is sorted the data could be printed as a time-series diagram, in which outliers and inliers are printed using different dot-shapes and/or colours, as illustrated in Figure 4.3. While this is not performing any binning, it may become problematic with a large number of rows. Since more rows will unavoidably extent the width of the image. This is a huge problem, since datasets for the purpose of the analysis should have a reasonable size, to make sure, the normal range of measurement values is covered and that the dataset contains some outliers. To achieve this we assume that at least 1000 rows, of sensor measurements, are needed. To render each point in the dataset at least one pixel of width is needed, this number probably is higher, to make the differentiation between inliers and outliers, by using different shapes or colours, possible. However, even with the, unrealistic, assumption of one pixel of image width per data row, a dataset with 1000 rows must be at least 1000 pixels wide. The size is no problem with 1000 rows, but it quickly becomes a problem with larger datasets such as 10,000 rows. Resulting in an image width of at least 10,000 pixels in width. Rendering this image on a screen, without scaling it down, therefore requires at least 10,000 pixels in width. Even on a 8K display, with a resolution of 7680 by 4320 pixels[8] cannot be used to completely display the resulting visualization, completely without reducing its size. Making the visualization impossible to render on almost any display, without shrinking the image size, which in turn may make some detected outliers invisible due to their small area used in the image.

The most suitable option for one dimensional visualization seems to be the histogram, since most of its problems can be mitigated, without compromising the accuracy of the images too much. But both the number line and the time series based visualization are suitable if one considers rendering them dynamically for example by using a web based charting library, making it possible to zoom within the diagram, in order to get a better view on important portions of the diagram.

4.3.2.3 Visualizing three dimensional inputs

For the visualization of contexts with three sensor inputs, an implementation similar to the two dimensional approach in a three-dimensional environment, is theoretically possible. The individual rows of the dataset are drawn in a three dimensional coordinate system and, as with the two-dimensional approach, are highlighted in colour based on the prediction or their label.

From a theoretical standpoint this is possible, but in practice there are huge challenges. Especially if the visualization should be saved in an image file, as it is is the case with both the two and one dimensional visualization types. To properly visualize the dataset, interaction using a virtual camera within the 3D environment, comparable to 3D modelling tools such as Blender[9] or three dimensional video games is necessary. To enable this, the visualization must be dynamically rendered, for example, within a special application or a special view in the web browser. Using third party dependencies such as a game engine or 3D rendering library, such as Unity[10] or Three.js[11] is good idea, to keep the mandatory effort at a minimum. Even with these libraries, mostly handling the 3D rendering,it is still very challenging and the usability of the visualization is questionable.

Without implementing the 3D visualisation, it is difficult to judge whether moving in three-dimensional space is possible enough to detect outliers and normal data. Whether and how useful the form of visualisation is needs to be investigated accordingly.

4.3.3 How to compare machine-learning based outlier detection methods?

In the previous sections, several options to make machine learning based outlier detection methods comparable, using visualizations and metrics have been introduced. The next step is, the implementation of a mechanism that allows the automated comparison of several outlier detection methods.

Generally speaking, the answer to this problem is very simple: Just run all outlier detection methods using the same input data. While this roughly describes the procedure, there are several important considerations that have to be taken into account in order to make such a comparison feasible:

[8]https://ihax.io/display-resolution-explained/
[9]https://www.blender.org/
[10]https://unity.com
[11]https://threejs.org/

1. Using the same dataset, in the same order, all the time is the most important aspect, since the measurements are only comparable if this is ensured. While this is relatively simple to achieve in many cases it can become a challenge if the data should be randomly shuffled. To ensure the outliers in the dataset are evenly distributed. For that, the randomization option has to be deterministic, most implementations of a random number generator, mostly pseudo-random number generators, support this by setting a seed value. The seed also has to be kept identical for every analysis and it has to be ensured, that the randomization procedure is identical, in all cases. Because, the seeded number generator will only produce the same output, if the operations executed, i.e., the number of calls to the random number generator are always identical. Another option, to get the dataset randomized could be the initial randomization, executed once on the dataset, before any of the outlier detection methods are invoked for analysis.
2. Since performance is compared, the environment of execution should be identical during the whole period while all of the outlier detection methods are analysed. On the one hand, this means that the analysis for one job should not be performed on multiple machines, ensuring that all outlier detection methods are executed on the same hardware. To make sure, the number of available processors and the available memory is always identical. On the other hand, it should also be ensured, that the system does not experience load from other processes during the analysis, to achieve that, the number of processes, apart from the analysis itself, should be kept as low as possible, if the most accurate performance comparisons should be achieved.
3. While the previous one is only mandatory if the user wants to achieve comparable performance measurements. The requirement that all parameters, as well as, the procedures used to perform the analysis should be kept identical for every one of the analysis. Is mandatory in all cases, not only to make sure the randomization works properly. But, to make sure the number of potential outside factors influencing the performance is kept to a minimum. The best option to achieve this, is the usage of a generic implementation, that only receives an instance of an interface implementing the corresponding outlier detection method.

The list of all outlier detection methods, on which an analysis has been performed on, should be ordered using a metric that describes it as best suited for the use case. In case the input data is labelled metrics like the accuracy or the F_1-score can be used for this. From observations it was determined, that the accuracy is sufficient for sorting the outlier detection methods in most cases, since a lower F_1-score often implies a lower accuracy. However, a mechanism to select the criteria to sort on is another option that could be considered.

4.4 Adding intermediate validation nodes to the modelling tool

The IoT Application Modelling Tool used as a basis, to embed the suggestion method, only supports sensors and actuators that act as a data source or a sink respectively. To perform any kind of intermediate processing a new type of nodes has to be introduced. We call this node type validator, since its main purpose in our work is the performing of validation / filtering of input data.

Unlike, actuators and sensors these validators are not a child type of an operator, since the deployment behaviour of them differs from actuators and sensors. One significant difference is the origin of the code that should be executed. Unlike operators the source code for the validation originates from another source, for example, the messaging engine or the IAMT backend itself, even though the flow of validation deployment is not implemented in this work, we will discuss multiple options for the execution of validation methods and where the validator implementation should be stored later.

Some examples for which the validator types can be used, include:

- Rule-based filters, such as an upper or a lower boundary filter.
- Machine-learning based outlier detection, the type this work focusses on.
- Transformation filters, for example a filter that transforms a measurement of volume from gallons to litres. Generally, such transforming filter operations are not the main purpose of the validator concept. However, with some adjustments mechanisms like a message transformer could be introduced.

Adding a validator to a model should work very similar to adding an operator. The modeller should just select it from a menu in order to place it on the modelling space. However unlike sensors and actuators, which usually only have outward facing and inward facing edges respectively a validator is intended to have both inward and outward facing edges.

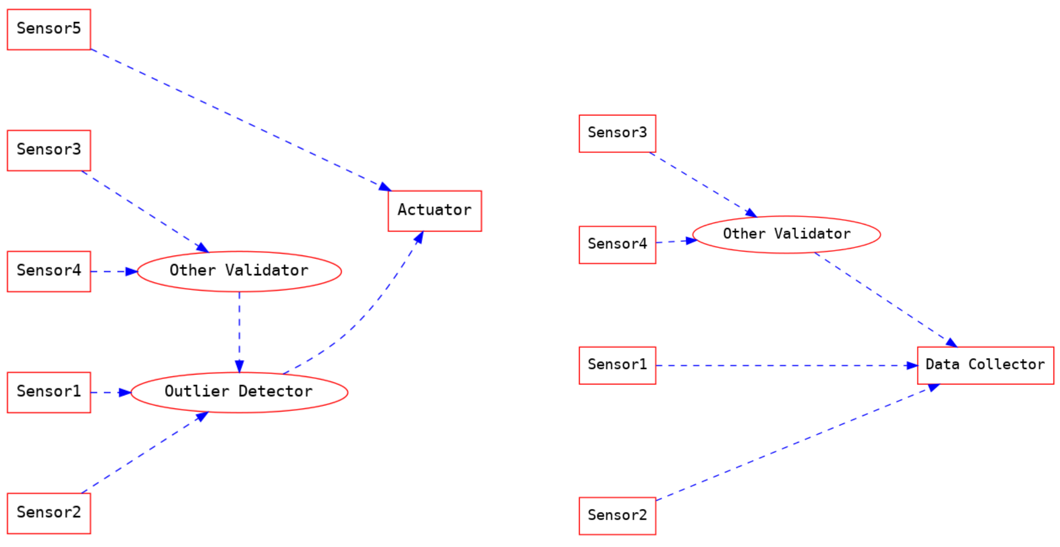
4.4.1 Collecting data by deploying the model partially

Suggesting an outlier detection method using this analysis method proposed above is possible in two cases (1) labelled data is available in a sufficient quantity or, (2) unlabelled data is available in a sufficient quantity. However, by combining the previously described analysis methods with the IoT Application Modelling Tool introduces a third mode of operation: Collecting unlabelled data and then performing the analysis.

To achieve this, the question which measurements from which nodes have to be collected in order to have all the mandatory data available to perform the analysis must be considered first. This can be achieved by, collecting all the measurement data from all direct and indirect inputs of the validation node. This collection could be done by creating a model, that only contains the nodes of interest and their edges and instead of the validation node, a special node is placed there with the task of collecting the data. When collecting the data must flow through the whole chain of nodes, for example if the model contains a Sensor (S_1) which is intended to send data to a validator (V_1) that then sends the data to an outlier detection validator (V_2), the data must from S_1 must still be passed through V_1 before collecting it. Determining the nodes that have to be included can be achieved by traversing the graph in reverse direction, starting at the node of interest.

As already mentioned, the outlier detection node must be replaced with a special operator, that collects the data into a database. Allowing the extraction of the data, once enough data has been collected.

To make the process understandable, an example model, shown in Figure 4.4 will be used as an illustrative example. In this particular case the *Outlier Detector* is the point of interest. In order to collect data, all nodes that are not a direct or indirect input of the *Outlier Detector* are not interesting-Since the data produced by other nodes, in this case *Sensor5* are not needed for suggesting an outlier



(a) The initial model

(b) The derived model for data collection for data to be used at the 'Outlier Detector' node

Figure 4.4: An example for a derived model

detection method at the point of interest. Therefore, the graph can be reduced to only include all inputs of the *Outlier Detector*, also the detector itself must be replaced with a special data collection node, that tells the deployment procedure to deploy a program that will collect the incoming data. This could be a separate program or a component that gets integrated into the messaging engine.

One approach to handle this approach using the IAMT is, by creating a new model, that only contains the relevant nodes and replaces the outlier detection validator with the collector node, assuming this is an actuator, which is one possibility, the node will be replaced with an operator, that is intended to perform the data collection.

Of course, this approach comes with limitations: It is only usable if the model does not contain any cycles before validation node of interest. Another limitation is the problem that, this process may only work in certain environments, without causing any damage, since the system is only in an observing state and will not perform any actuations. This must be considered, and may prevent the usage of this approach in some use-cases. It also cannot produce labelled data, making the process semi-automatic. Because, the modeller still has to select the best suited outlier detection method manually. If this process could also be automated, the complete suggestion could happen automatically.

4.4.2 Introducing the outlier detection wizard

At the current stage, we identified an approach to find the best suited outlier detection method based on present data. The next task is the transformation of this (theoretical) concept into something that is part of the IoT Application Modelling Tool.

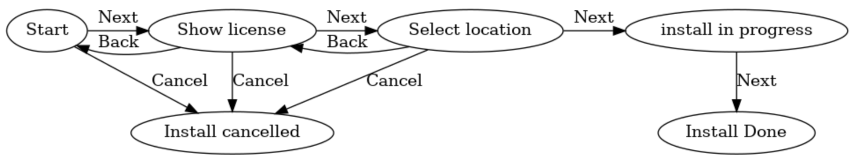


Figure 4.5: Graph illustrating the steps (states) and the transitions between them

With the main goal of making the selection of the best suited outlier detection method as simple as possible for the user, we decided to build a configuration wizard, that will guide the modeller through the process, by asking for mandatory information, like the mode of operation and the data at the point where it is needed. For this we took inspiration from several sources, mainly installation wizards for both desktop applications like Microsoft Office and web applications such as NextCloud[12], as well as other step based processes like checkout procedures in online shops. An installation wizard is generally build up into multiple steps, like accepting the license agreement, choosing the installation location, setting up the database connection and creating an administrator user account. These steps build on each other and the flow of a wizard can be drawn as a state machine, illustrating the possible transitions from any given step.

Such a simple state machine is shown in Figure 4.5. The states can be grouped into three stages: (1) The **configuration stage**, where the user agrees to the license and selects the install location, (2) the **processing stage**, where the installer performs its actions and the progress of these actions are shown to the user, and lastly (3) the **result stage**, where the results are presented to the user. In the case of such an installer this will probably only inform the user that the install was successful. During the first stage, the user can also go back, to allow the user alteration of the inputs. In the second stage the installer performs the installation, here cancellation is not possible any more in our example, whereas the cancellation was still possible before entering this stage. Once the installer is done a message is shown to the user informing him that the install is done.

In our wizard concept the first and second step will be identical, with regard to their main goals, getting mandatory inputs and showing progress respectively. The main difference lies in the third stage, here display the results of the analysis will be displayed, by showing which outlier detection methods suit the use case best. For example, by showing the resulting accuracy or the visualizations. At this stage the user also has to choose the desired outlier detection method.

After roughly identifying how the process within the analysis will be displayed to the user in the IoT Application Modelling Tool, the following options had to be investigated further*:

1. How does the user get to the wizard?
2. How are the results of the wizard stored and displayed in the model?
3. What inputs do we need from the user to perform the analysis?
4. In which sub steps will the wizard be divided and how will they look?

The initial idea how the wizard should look like has been described in wireframe mock ups. Most of these mock ups can be found in the appendix (Appendix A.2). However, the most important ones are within this section. The following part will cross-reference to the wireframes accordingly.

[12]https://nextcloud.com/

4.4.2.1 How does the user get to the wizard?

To embed the wizard into the application as best as possible, we decided that the user will select a validator node from the validator menu. The user can then use this node to create the connections to other nodes (See Figure A.1). Before opening the configuration wizard by expanding the detail view of the node and clicking a button that will open the wizard as a modal dialogue (See Figure A.2). While the model contains such unconfigured nodes, the instantiate button should be disabled, since the model still has to be configured before deploying it.

4.4.2.2 How are the results of the wizard stored and displayed in the model?

In the modelling tool this should just transform the unconfigured node into a configured one, that contains the selected method, as well as the dataset (See Figure A.7). To make changes in the decision possible, one should also be able to reopen the configuration wizard somehow.

4.4.2.3 What inputs do we need from the user to perform the analysis?

The two main inputs the suggestion method needs are: the mode of operation and the data-sample, however the option to add new parameters was still important, since the suggestion method relies several parameters that may be altered. The following parameters were taken into account: (1) The percentage of rows that should be used for testing, (2) a list of outlier detection methods that should be excluded, (3) a boolean option to enable randomization of the dataset before performing analysis and (4) a seed for use with the randomization, if it is enabled.

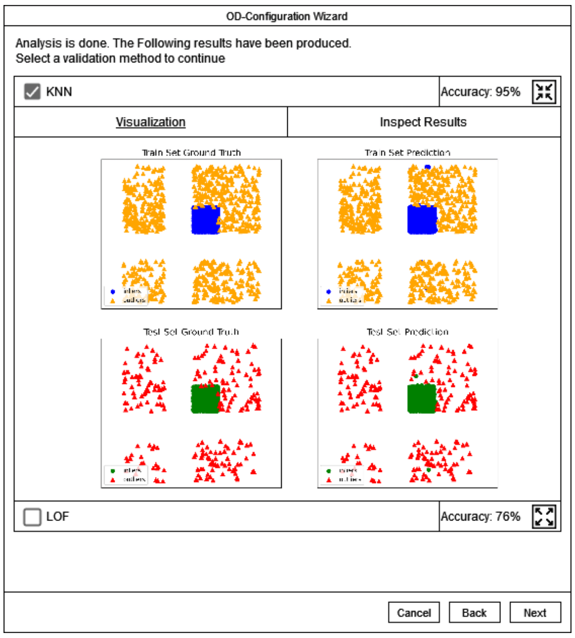
4.4.2.3.1 In which sub steps will the wizard be divided and how will they look? The wizard will be built out of the following steps:

1. The wizard starts in the **Opening Screen** that has the main goal of setting the mode of operation for the wizard. Some informational texts, for example, describing the approach and showing the supported outlier detection methods are included in this step. The wireframe depicted in Figure A.3 only includes the main goal of selecting the mode of operation. In our implementation the wizard only allows you to proceed if data is already present. The next steps for the other two options, *having no data but data collection is possible* and *having no data and data collection is not possible* are not defined here, since they will not be implemented in this work. Some ideas on the implementation of the data collection based step is discussed in Section 8.1.3.
2. After one of the two modes of operation have been selected, the modeller can proceed into the **Data Upload** step. Here the user has to choose a csv file, in a defined schema, containing the dataset used for performing the analysis. This step is also intended to configure the additional parameters. However, they have not been added to the wireframe depicted in Figure A.4.
3. After the data has been uploaded the wizard continues into the **Input data review** step, here the modeller can review the uploaded data. To make sure it has been parsed properly. Because, the dataset will usually contains hundreds or thousands of rows the contents are displayed in pages, using a paginated view, to improve the performance of the frontend application.

Another well performing option that is potentially suitable for rendering such a large list are virtual scroll lists that could be provided by the Angular Material CDK [Goo23]. The wireframe, based on the paginator, or this step is shown in Figure A.5.

4. The **Processing** step is self-explanatory, here the backend has started performing the analysis and the task of the frontend is mainly displaying the current progress by means of a progress bar. The wireframe of this step is depicted in Figure A.6.

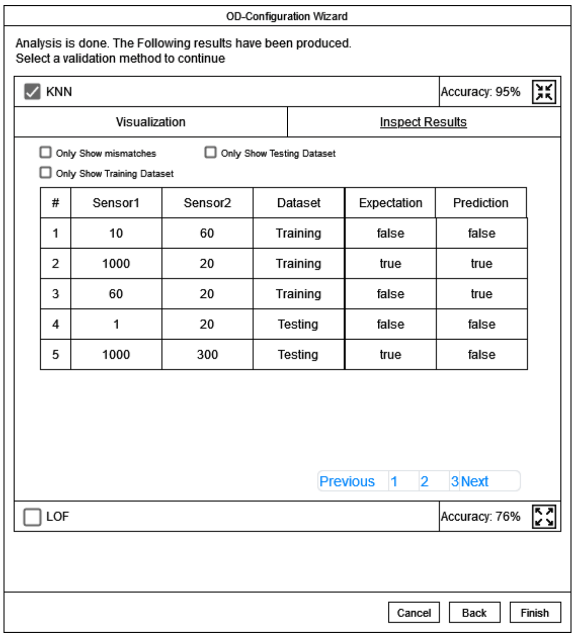
5. The last and most important step is the **Result Review** step. Here the results of the analysis will be presented to the user. The results are presented in a list, showing the name of the method that has been executed as well as the computed accuracy (in case of the supervised mode of operation). This list is either sorted by the accuracy or the name of the outlier detection algorithm, if the accuracy is not available.
By clicking on an outlier detection method, a detail view will be shown, that will show the visualization of the result (if available) and the raw output data, containing each row in the input dataset, similar to the third step (input data review) the list is going to be using pagination to ensure the application performs well. The wireframes for both the visualization and raw data view are depicted in Figure 4.6.
The selection of the algorithm is done by selecting its corresponding checkbox and pressing the next button afterwards. Configuration of the node is done at this point and the model may be instantiated now, if there is no other unconfigured node within it.



OD-Configuration Wizard

Analysis is done. The Following results have been produced.
Select a validation method to continue

(a) Result view with data visualization



OD-Configuration Wizard

Analysis is done. The Following results have been produced.
Select a validation method to continue

#	Sensor1	Sensor2	Dataset	Expectation	Prediction
1	10	60	Training	false	false
2	1000	20	Training	true	true
3	60	20	Training	false	true
4	1	20	Testing	false	false
5	1000	300	Testing	true	false

(b) Result view with raw data view

Figure 4.6: Wireframe of the results step of the wizard.

4.5 Choosing and implementing outlier-detection methods

There are many different types of ML-based outlier detection methods, that are suitable for use with the analysis methodology proposed beforehand. However, performing the analysis on all of the methods can be very time consuming. Because, some of the selected outlier detection methods may take minutes to perform their analysis each. As a result, a subset of all available outlier detection methods must be chosen. Some factors that help limiting the set of outlier detection methods are the type of data that has to be analysed, the environment the analysis is performed in and the degree of supervision of an outlier detection method.

However, as this work only proposes a mechanism to determine which outlier detection method is best suited for any given dataset. It must be ensured that new outlier detection methods can be added reasonably easily to either add new methodologies proposed in new research or outlier detection methods that already exist but have not yet been added to the set of supported outlier detection methods.

The following machine learning based outlier detection methods have been selected to be part of the initial set of supported outlier detection methods:

- *k*-nearest Neighbors (kNN) [AP02; RRS00]
- Local outlier factor (LOF) [BKNS00]
- Clustering-Based Local Outlier Factor (CBLOF) [HXD03]
- Isolation Forest (IForest) [LTZ08; LTZ12]
- Angle based outlier detection (ABOD) [KSZ08]
- Connectivity-Based Outlier Factor (COF) [TCFC02]
- Principal Component Analysis (PCA) [SCSC03]
- Deep One-Class Classification (DeepSVDD) [RVG+18]

All these methods do not need labelled data and therefore use unsupervised learning. In order to also have an outlier detection method that is not based on unsupervised learning Extreme Boosting Based Outlier Detection (XGBOD) [ZH18] has been chosen, since there was an implementation of this one on hand. In fact XGBOD uses a semi-supervised learning approach. However, since the proposed wizard does not support semi-supervised methods it will be used as a supervised-learning based outlier detection method.

The unsupervised approaches have been selected, because of previous research, for example, by Jiang et al [JHSG+20] and Al et al. [AMM+21] they investigated the usability of different outlier detection approaches in the Internet of Things context that use several of the previously mentioned machine-learning based outlier detection methods as a basis, including, *k*-nearest Neighbour, Local Outlier Factor and Principal Component Analysis.

4.6 Investigating the implementation of validation capabilities in the tool stack

The tool stack in its current state is simply unaware of the validator concept introduced in Section 4.4, considerations on how the mandatory abilities can be implemented have to be investigated, in order to implement the process of validation. For this, most of the tools have to be modified. This section will investigate design challenges one may face when implementing the instantiation of validation nodes.

To achieve this, the following modifications have been identified:

- The modelling tool has to be adjusted to allow the modelling of validation between sensors and actuators, this has already been covered in Section 4.4.
- It is mandatory to find out where, i.e., on which device, the validation should get executed on. The general question here is: should the validation for the whole data flow be executed on one (central) Device or should it be done decentralized, i.e., on the output of a sensor or the input of an actuator.
- After the approach for the validation is selected the ME-Configurator has to be able to handle the deployment the validation.

This section covers an architectural investigation on how the validation should be implemented, mainly focusing on the second challenge described above.

4.6.1 Identifying modelling cases for validation

At the basis, we identified four types of modelling cases that describe how inputs and outputs are related to a validation node (validator). Such a validation node does not have to implement an outlier detection method, instead it can be considered a more abstract concept representing intermediate nodes, comparable to filters in the pipes and filters pattern [MC22]. The taxonomy of the modelling cases is similar to relationships between entities in relational databases.

The following paragraphs will take an abstract look on the task of one validator. For this we will call incoming nodes inputs and outgoing nodes outputs. Although this definition is intuitive, it is important to note that an input or an output can be not only an operator, but also another, chained, validator. In this first stage of the investigation these cases are considered identical in Section 4.6.2. The difference between validators and operators in both a centralized and decentralized validation approach are investigated later.

One to One ($1:1$) The simplest modelling case is the one-to-one relationship. Here the validator performs the validation based upon one input, produced by the output of a sensor or by another validator. In this case, the validation is carried out on the basis of an input, which, in the case of a positive check by the validator, is sent on to an output. An example for this is a filter that checks if the value is in between a lower and an upper boundary.

Many to One (n : 1) In such cases the validation may not be based upon just one input. However the node still only produces one output. For example, a validator may want ensure the reading of some sensors is accurate to do this two sensors of the same kind may be used. In case the values differ severely the validator may not produce an output to prevent the activation of the actuator.

One to Many (1 : n) and Many to Many (n : m) In some cases the validator may not only send the output to one target node, like we assumed above. However, sometimes a validator may have to send its result to multiple outputs. For our purpose it is assumed, that every output will receive the same message and therefore the output will only produce one output value. If a validator may produce to different outputs, it may be decomposed into two separate validators.

4.6.2 Investigating central vs. decental validation

The Messaging Engine and the associated lifecycle is focussed around decentralization [DH20]. Therefore, the initial idea was designing the validation in a decentralized manner. However, centralized validation was also considered an option, due to the reduced effort in more complex modelling scenario.

To implement validation in the model there are three different approaches, all having pros and cons that will be evaluated next. The three most reasonably approaches are:

1. The fully centralized approach, here all validation for the whole model is done on one special device, called the validation device. All messages that go through a validator are sent to this component for validation.
2. The fully decentralized approach with validation done on the existing devices of the operators implementing validation without further adding any new devices or (shadowed) operators to the model. With this approach the user generally is not in charge to find out where the validation should be implemented. This instead is decided by the model, by applying rules to it to determine where the validation should be executed.
3. Validators get similar treatment to operators, requiring the user to select the device on which the validation should be deployed.

In regards to the degree of automation and the reduction of complexity for the modeller, the full decentralized approach is the most favoured, however it comes with some serious drawbacks that have to be considered when implementing validation.

4.6.3 Investigating fully decentralized validation on the operator device

For the fully decentralized approach the following basic principles were defined, with the main goal to prevent violations of them where possible. In order to keep the mandatory computation resources as low as possible.

- The introduction of a new deployment unit, i.e., a piece of code that is deployed on a device, like an operator, should be prevented where possible.

- Duplicate execution of validators should be avoided, to ensure all outputs that use the validator get the same value. Depending on the type of validator duplicate execution may produce different results if the implementation is non-deterministic, as it may be the case in some probabilistic code. Executing the same task multiple times also results in unnecessary use of computation resources.

The development of an approach that does not violate these two principles can become very complicated, as shown in the following. First looking at the point of execution in the data flow, i.e., where the validation should be executed. Tor that the four different relation types will be discussed.

- Where **One to One** relationships are executed in a decentralized execution environment does not matter, since both the source and target node will have the inputs available. In this case the decision can be delegated to the person instantiating the model. The validator is then considered a deployment unit in combination with its selected operator.
- The choice of devices gets smaller if a second input is added to the validator, then the modelling case turns into a **Many to One** relationship. As validation should be done at a node where all inputs are present the target node should be chosen to execute the validation.
- In **One to Many** and **Many to Many** relationships there is no node at which all inputs are present, because of that different ideas, like introducing a new node, have to be investigated. This will be discussed further in Section 4.6.3.1.

4.6.3.1 Handling one-to-many and many-to-many modelling relationships

In **one-to-many** and **many-to-many** cases, there is no node through which all inputs and outputs flow, as a result, the rules described in the previous section are either difficult to comply with or they must be violated.

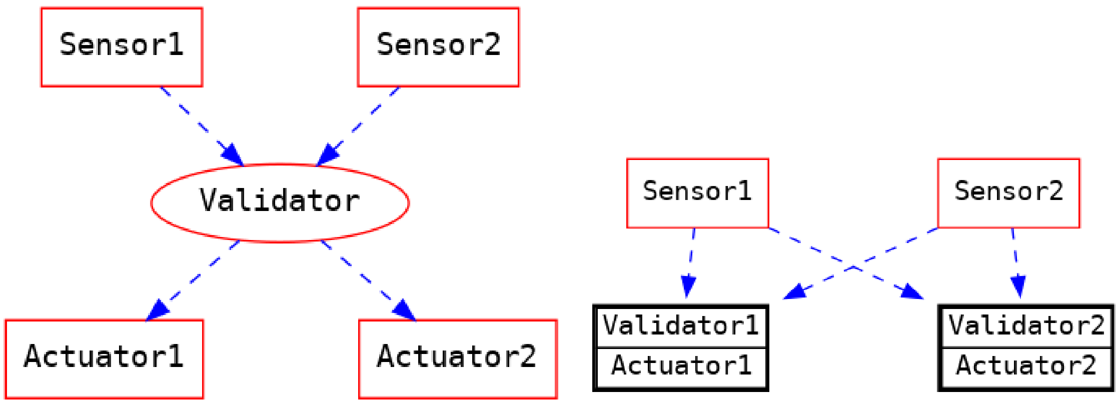


Figure 4.7: The decomposition of a many-to-many validator (left) to two many-to-one validators (right)

If these potential violations of these rules are accepted, the following options are some possible solutions to the matter.

- The validator is deployed on every target node, by decomposing a many to many validator into multiple many to one validators, which can be deployed on the device of the output, as discussed in the previous section.
  A visual illustration of this process is shown in Figure 4.7, here a 2 : 2 validator, named `Validator` is internally transformed into two many to one validators `Validator (1)` and `Validator (2)`.
  One problem of this approach, is the violation of the previously described double execution rule, which may cause issues depending on the implementation of the validator. Before we take a deeper look at the problems of this approach we will cover some potential benefits, these include:
  - The system is more fault proof, as no new single point of failure is introduced, in comparison to the approaches we will present below.
  - This approach can be used to model one-to-many and many-to-many validation relationships even though they are not even supported by the system itself, only the many to one case must be supported.
  - By following this approach the system stays fully decentralized.

  On the other hand this approach has a lot of arguments against it, including:
  - The, already mentioned, possibility to produce inconsistent behaviour if the implementation is non-deterministic.
  - Long-running or compute-intensive processes are executed multiple times, increasing resource requirements. Especially, on IoT devices, known for having limited computing and memory resources, this can become a significant problem.
  - This approach has a huge networking overhead when working with many inputs and/or outputs. Generally the number of packets in the system that occur for every measurement cycle (for example: every 30 seconds), depend on the number of inputs and outputs. If we assume that we have a model with many inputs and many outputs, for example 100 each, connected by a validator then the number of packets sent by this approach is $100 * 100 = 10000$ since every sensor of the 100 existing ones has to send its reading to every actuator ($n * m$).
  - With this increased network traffic, the probability of having packet loss increases, which is another cause for inconsistent behaviour within the system.
    Looking at the decomposed graph illustrated in Figure 4.7 (right figure), it can be observed, that both of the decomposed validators are executed on their attached actuators, the packets originating from `Sensor1` and `Sensor2` are directly sent to both actuators. assuming that the message from `Sensor1` to `Actuator2` is getting lost, then the value produced by the attached validator (`Validator (2)`) may differ from the one produced by the other one or it may not even produce any output due to missing data producing inconsistent results.
- The validator is deployed separately on a device, that is chosen by the user at the time of instantiation, causing a violation of the rule that no additional node should be introduced. Apart from the violation this approach does have a lot of benefits, including:

  - The fact that this approach is applicable to every relationship we described, potentially allowing the simple implementation of decentralized validation, with the drawback that new nodes are introduced.
  - If an implementation of a validator is resource intensive, this option is the best one for fully decentralized validation, since someone, probably the person modelling the flow, has the ability to chose the device on which the validator should be executed and if this person is aware of the resource requirements a more powerful device can be selected.
  - The network traffic is reduced, in comparison to the approach described previously, since sensor readings are only sent to one node instead of many ones. In contrast to the previous option this one only needs $100 + 100 = 200$ packets ($n + m$) for the aforementioned example with 100 sensors and actuators each, since every sensor sends its reading to only one node being the validator, resulting in a total of 100 packets. The validator then sends its result to the connected actuators which also needs 100 packets. However there are some limitations to this argument: 1. the effect is less significant when looking at smaller numbers of inputs and outputs, and 2. the effect is slightly reversed in the One to Many case here the decomposition needs n packets and this approach needs $n + 1$ packets

- The validator is deployed with one of the outputs as one deployment unit chosen at random, by the user or by the modelling tool, for example based on conditions, like the number of available processor cores, this validator will then send the validated result to the other target nodes, as illustrated in Figure 4.8.
  Generally, one can consider this approach to be a combination of the two previous approaches instead of introducing a new computation unit an existing one is expanded that now also has to act as a validator. In contrast, to the approaches described before this one does not introduce a new deployment unit nor does it cause duplicate execution of the validator. But this approach also has some drawbacks, including the following ones.
  - Just like the first approach, this one also introduces a new single point of failure.
  - Computation power may be limited, since the validation is done on an IoT device, however this is a common drawback with decentral validation, affecting every approach that combines the validation and an operator into a deployment unit.

4.6.3.2 Handling chained validators

In order to follow the pipes and filters concept, validators must also support chaining of validators, i.e., the inputs of a validator may be operators or another validator. Implementing chaining is especially challenging when working in a fully decentralized environment, following the two rules of avoiding the introduction of new computation units and the avoidance of multiple executions for the same sensor value.

Generally chaining of validators can be mathematically expressed as a composition of multiple functions, for example, $C(B(A(x)))$ for the validator flow illustrated in Figure 4.9. Based on this idea, one option to select the device for execution could be the logical combination of all three validators into one, with this idea the three validators A, B and C into one ($V_{ABC}(x) = C(B(A(x)))$).

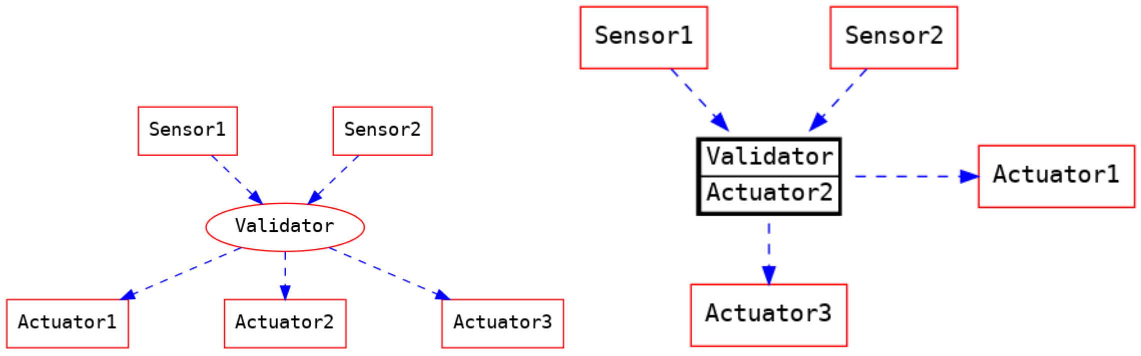


Figure 4.8: Visual illustration of the the per device mappings when running one-to-many and many-to-many validation on an actuator. The left image depicts the modelled many-to-many modelling case and the right one displays how this model gets deployed, i.e. every node represents one device

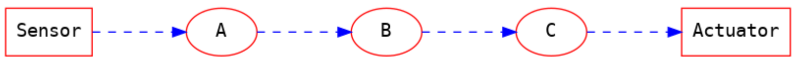
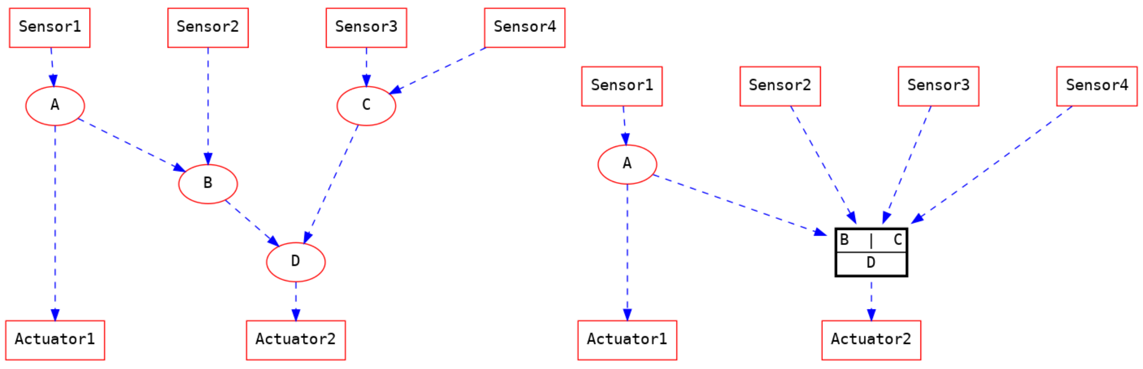


Figure 4.9: A chain of one-to-one validators

Once they are logically combined, the rules for running a single validator in a one-to-one relationship can be applied.

A similar procedure can also be applied to more complex chains of validators, that point to one target node, if its points to multiple nodes the graph is more complex, for that the chain must be split into several parts.



(a) Potential model with complex chain **(b)** Technical decomposition of the model, before selecting the target operator for deployment.

Figure 4.10: Potential topology with more complex chaining and its composition

As illustrated in Figure 4.10 the validators in the illustrated model cannot be combined into one validator. Instead, it can only be split into two separate ones. Because, the output of validator *A* is also consumed by *Actuator*1. From this example some rules can be derived: (1) Validators can be combined into one, if all source validators only point to another validator, as shown by the validators *B*, *C* and *D*, in Figure 4.10, since they turn into one combined validator *BCD* and (2) if a validator points to at least one operator this validator should be considered the end of a validator chain.

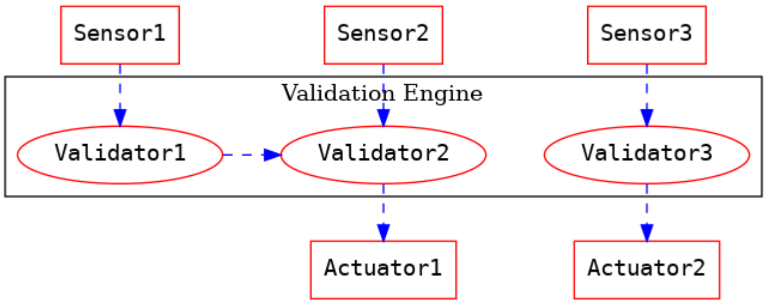


Figure 4.11: A sample model to illustrate central validation and the responsibility of the Validation Engine component

Following these rules the aforementioned graph can be decomposed into two validators, at first validator A, which has to stay as one. As it is, targeting an operator and the combined validator BCD that internally composes the following composed function: $BCD(A, Sensor2, Sensor3, Sensor4) = D(B(A, Sensor2), C(Sensor3, Sensor4))$.

After the validators have been combined into logical units, the target device on which the validator should be deployed can be determined. For that the same rules, as for non-composed validators apply. In the case shown in Figure 4.10 the validator A should be executed on the device of $Sensor1$, representing a one-to-many modelling scenario and the composed validator should be executed on the device of $Acutator2$, as this represents a many-to-one modelling scenario.

4.6.4 Investigating centralized validation

Before looking at the pros and cons of centralized validation, the term centralized validation in the context of this work will be defined. The centralized validation implements the complete validation for the whole model on one device, that runs a special application, called the validation engine.

Figure 4.11 illustrates the way this work defines central validation. Every Validator is executed on the Validation Engine a single component responsible for performing the validation in the given model. The device on which the validation engine is deployed is chosen by the user at the time of deployment.

Many of the problems we encountered in the space of decentralized validation are not a problem in a centralized approach, including the following ones.

- Where the validation should be executed is always clear, even if the model becomes very complex, for example, by including many chained validators.
- The validation is less constrained in regards to the resource usage, because the device used for validation has to be selected during the instantiation of the model. Here we could inform the user about the fact that the validation is resource intensive and therefore recommend selecting a more powerful device if the user wants to deploy on a low performing machine.
- The same validation is not executed twice, unless this has been explicitly modelled by the user, reducing the possibility of unexpected behaviour.

- The network traffic is not significantly increased and is on a comparable level to the decentral approach, with a new deployment unit for every validator. Theoretically it may be lower than that case, depending on the model. Because, chained validators could be handled as internal traffic by the centralized option.
- The deployment on the devices is much simpler. Since the code for the validation only has to be deployed once, however other parts of the model still have to be deployed decentralized like the Messaging Engine, making this argument almost irrelevant.

On the other hand the central approach also has one huge drawbacks, as it is a new potential single point of failure that does not only disable a portion of the model but the whole model, this of course also is one of the huge benefits of doing these tasks completely decentralized.

4.6.5 Investigating device selection by the modeller

Another option for the implementation of validation is the most intuitive, that on the other hand, requires more user input in comparison to the other two options. In this mode the modeller always selects the device on which the validation should be executed.

Without having to consider the aforementioned challenges, the effort of implementation for both the modelling tool and also the tool stack used to deploy the complete model is less complex.

As a compromise a mix of the fully decentralized approach and this one can be considered, in that case the system will not decide on the validation device, but will instead make a suggestion where the validator should be deployed. Therefore, the process can still be automated, but it offers the maximum flexibility for the modeller. In the case that it is mandatory to change the device, based on conditions the modelling tool is not aware of, for example, if unmeasurable constraints, such as the physical location of the device or other quality-of-service requirements.

Depending on the choice of the user the model may either be deployed in a more centralized or more decentralized manner.

4.6.6 What is the best option to implement validation

After the three potential options have been discussed, the best suited solution for the device selection is to leave the choice at the user, however the modelling tool should still be able to make suggestions in an automated manner in most cases. But by only giving suggestions it is no problem if a good suggestion cannot be made, in case the model context is very complicated.

With some optimizations, the combination can be used to reduce, the mandatory number of packets to send to other nodes and also the number of instances of which the validator may be implemented. Some ideas, how this could be handled will be discussed next.

4.7 Investigating the implementation of the Validation Engine

As previously discussed leaving the choice of the device to deploy the validator on to the used, with some assistance by the application is the most reasonable approach. In order to implement validation using this approach, a new component, that is either a separate application or an extension of the Messaging Engine is introduced, with the main responsibility of performing the validation, meaning that the validation engine must perform the following task for every instance of the validator.

Including the Validation component within the messaging engine, is the simplest option, because this way, it is made sure that running validation is possible on every device that runs the messaging engine. In case a device does only have one or more validator attached and no operators selected for execution, it must be ensured an instance of the validation-enabled messaging engine is deployed on the node.

During deployment (instantiation) it should be made sure, that only one instance of the messaging engine is deployed on a device, to reduce the resource usage of the application on the device, since all messages and validations are handled within one central process, potentially invoking subprocess if mandatory.

In order, to reduce the network traffic some optimizations in this extended messaging engine should be implemented. Assuming there are two nodes that should be instantiated on the same device, one validator, that will transmit its results to an operator, the engine should not send the values to the network. Instead, it should internally pass the message from the validator to the operator.

To further keep the memory resources needed, as low as possible a mechanism to only instantiate the validators if they are assigned to the specific device, because only then the instantiated version will be used.

4.8 Investigating changes in the workflow to deploy a model

How validation must be deployed is depending on the validation approach that is chosen, the main commonality of all options is that the ME-Configurator and the Messaging engine in all cases have to be modified, in case of the decentralized approaches more significantly than in the centralized approach.

In all cases the input payload of the ME-Configurator must be adjusted to consume information about how and where the validation should be performed. However, just like the differences with the overall workflow this differs by the approach of implementation chosen.

The simplest approach to implement the validation is the centralized one, where the ME-Configurator has to receive a list of all validators including their connections as well as the location (Device) the validation engine should be deployed on, and all the other parameters required for instantiation of the model. At the time of deployment the ME-Configurator, just hast to make sure the centralized validation-engine is deployed on the selected device and that the deployed instance of the validation engine has received a configuration file telling the validation engine which validation tasks it has to perform and on what endpoints it has to listen for input data.

In both of the proposed decentralized options the validation component is expected to be be a part, or an extension, of the messaging engine, that can then in turn handle the validation. In both of these cases the deployment workflow must not get modified significantly, since the ME-Configurator just has to roll out the validation aware messaging engine on the specified nodes, with the appropriate configuration that then also contains information about the validation.

The main changes, in that case, have to be made in the messaging engine, where the validation is either added as an extension or a separate application, that gets deployed if needed. These changes, again differ based upon the option, that has been chosen in regards to decentralization. However, the general procedure to initialize the outlier detection based validator is always identical, even when implementing the centralized option. To achieve this, the validation component must receive a configuration, containing the location of either the input data, to retrain the model, or the serialized version of the trained model, that has been created during the analysis process.

5 Implementation of the suggestion wizard

This chapter covers the technical details, behind the implementation of the wizard that has been proposed in the previous chapter. At first, the relevant components of the tool stack are described on a more technical level. Next, validator nodes will be introduced into the modelling tool. After, that the technical details on the implementation of the wizard in both the backend and the frontend, before finally investigating options to randomly generate sensor data, that can be used to test the functionality of the wizard.

5.1 The tool stack on the technical level

The tools and their functions within the tool stack have already been discussed. However, before going into the technical details of the implementation of the outlier detection wizard a brief introduction on how these tools are implemented, will be provided, in order to better understand the reasoning behind certain decisions. The Messaging Engine and the ME-Configurator are not covered here, since they are not mandatory for the implementation of the wizard, interaction with these is mandatory once the instantiation should be implemented.

5.1.1 MBP - Multi-purpose Binding and Provisioning Platform

The MBP is an open source tool[1] with the purpose to ease the deployment, monitoring and management of IoT-Devices. It provides a centralized option to orchestrate IoT Devices. From a technical standpoint the MBP is a Spring Framework based Java application running either as a Java Servlet running on a Servlet server, such as Apache Tomcat[2], or as a stand-alone Java Executable (`.jar` file). The application state is stored in a MongoDB[3] for communication between IoT devices and the MBP the MQTT protocol is used. To use MQTT a broker software is needed, the broker Mosquitto[4] is used to serve this purpose [FHS+20]. Running the application is usually done using Docker[5] Containers and Docker Compose[6].

[1]https://github.com/IPVS-AS/MBP
[2]https://tomcat.apache.org/
[3]https://www.mongodb.com/
[4]https://mosquitto.org/
[5]https://www.docker.com/
[6]https://github.com/docker/compose

5.1.2 IoT Application Modelling Tool

The main focus of this work is the IoT Application Modelling Tool, it is the basis for the previously proposed outlier detection method suggestion wizard, this tool consists out of three components, all using different programming languages and frameworks: (1) the backend, that is used to persist the models created in a document based database (MongoDB), it is implemented in Kotlin[7] and uses the Spring Framework[8] as a basis, (2) the frontend, implemented as an Angular[9] application, this is the main part of the application, it gets loaded in the users browser and accesses the stored models, and (3) lastly the gateway, it acts as a middleware implemented in Node.js[10] between the backend and the frontend and is used to perform authentication using the user and permission management from the MBP.

In order to run the application tools like Docker and Docker Compose are used, to ease the process of deployment on any device.

5.1.2.1 Looking at the data model

The data model of the modelling tool is relatively simple, on a persistence level there is only one document, for every Model instance: the `Model` class. It stores all information about one specific model, like its identifier and name, as well as the Nodes contained by it. The array of the contained Node objects contain the whole model, apart from the model name and the model identifier. Each node stores an identifier, that is defined by the frontend application and must be unique in the context of the model, a name, its type and a list of the identifiers of other nodes that the node connects to (outbound edges).
A node also stores data about the operator it is associated with, this data is retrieved and copied from the MBP. The copy will contain the values of the parameters associated with the operator, making it possible to use them for instantiation later.

The data model of the modelling tool is illustrated in Figure 5.1 as an UML class diagram. In its implementation there are more attributes, that describe the visual state of the model, for example, if a node should be extended or the icon that is associated with it. However, these have been omitted in these class diagrams, to prevent the class diagram from becoming bloated with attributes, that do not serve a purpose for the overall functionality. All of these fields only have impact on how the nodes are visualized in the frontend.

5.2 Adding validators to the modelling tool

One of the most important things that have to be implemented are the validator nodes, that act as middleware nodes in the data flow between sensors and actuators. These nodes are referred to as validators and they are mandatory in order to integrate validation, in our specific case outlier detection, into the IoT Application Modelling Tool.

[7]https://kotlinlang.org/
[8]https://spring.io/
[9]https://angular.io/
[10]https://nodejs.org/en/

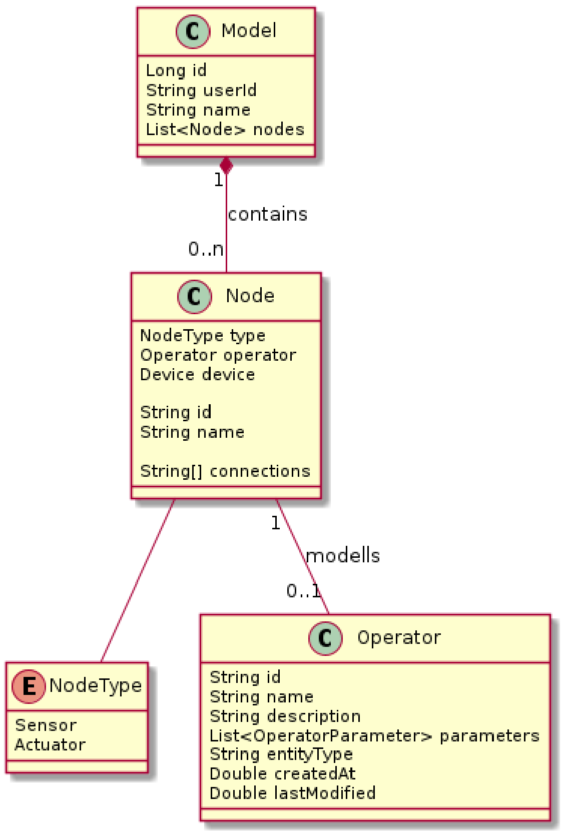


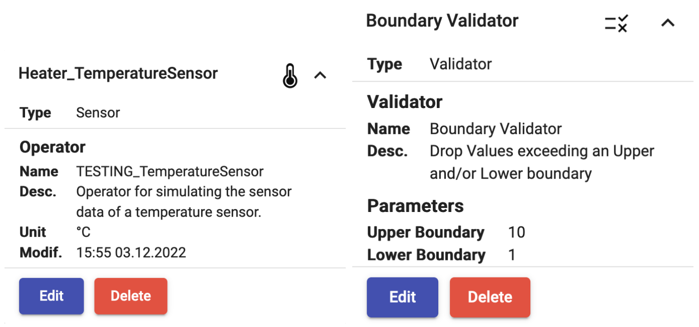
Figure 5.1: UML class diagram illustrating the data model of the IoT Application Modelling Tool

In order to make validators addable to the modelling space, the sidebar used to add operators has been extended to have a separate tab, containing all available validators. These validator types should be fetched from an external data source that is aware of what is supported by the rest of the system. However, as this work only implements the suggestion wizard without handling the instantiation, this has been omitted- Instead this list, which should only contain the type of the suggestion wizard, is provided by a statically implemented list within the frontend.

Placing a validation node is also similar to the process used for placing an operator, there are some minor differences, including the lack of (sub)-type selection, since the implementation does not differentiate between the types of validators. Unlike operators, where there are sensors and actuators.

Once the validator has ben instantiated, it is displayed in the modelling space, just like an operator node. As illustrated in Figure 5.2 the major differences are some missing attributes in the validator, like the unit and the modification time. These attributes are omitted, since they are not used in a validator.

Implementing this feature also made some changes in the data model mandatory. The most important change is the creation of a new parent class for the definition of operator and validator types. This class, called ModelNode has the same attributes as the Operator class in the initial data model, illustrated in Figure 5.1. The operator class and the newly introduced validator class inherit from this class, these classes do not introduce any new attributes and are only used to make differentiation



Heater_TemperatureSensor

Type Sensor

Operator

Name TESTING_TemperatureSensor
Desc. Operator for simulating the sensor data of a temperature sensor.
Unit °C
Modif. 15:55 03.12.2022

Edit Delete

Boundary Validator

Type Validator

Validator

Name Boundary Validator
Desc. Drop Values exceeding an Upper and/or Lower boundary

Parameters

Upper Boundary 10
Lower Boundary 1

Edit Delete

Figure 5.2: An expanded sensor node (left) in comparison to an expanded validator node (right)

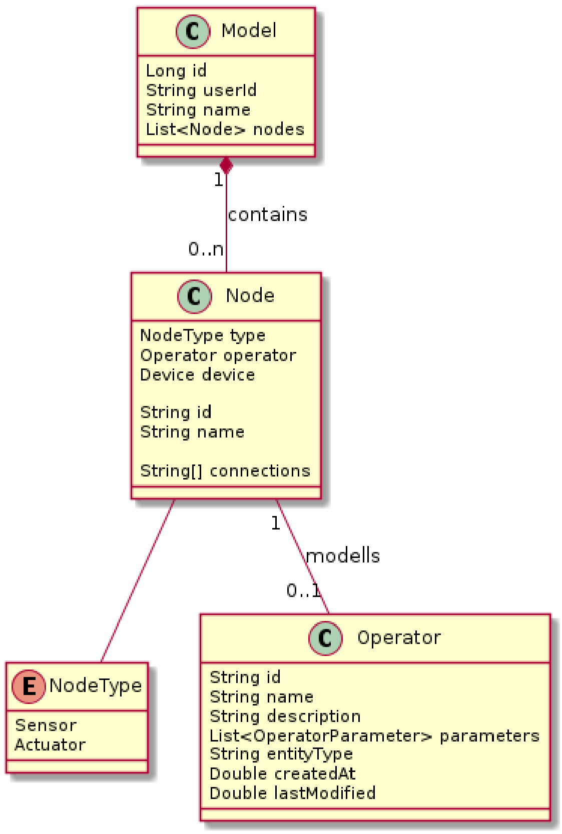


Figure 5.3: UML class diagram illustrating the data model of the IoT Application Modelling Tool after the validator nodes have been implemented

between them easier. Two further modifications of the model took place: 1. A Validator object has been added to the NodeType enum, used for simple differentiation between the types, and 2. An attribute to store the validator object associated to the node has been added to the node class. By design this validator attribute and the operator attribute in the Node class mutually exclude themselves, meaning, that only one of the should be set while the other one is `null`.

The changes to the model are also illustrated in a class diagram, shown in Figure 5.3.

5.3 Implementing the Backend

Before starting the implementation of the frontend of the wizard, we had to investigate how the actual analysis will be implemented. From a design perspective there are multiple options to implement this. At first, it was mandatory to decide where the analysis should be implemented, either in the frontend or the backend. Followed by the decision how the outlier detection methods should be implemented.

To make sure the analysis and the validation can use the same implementation of the outlier detection algorithms, it was mandatory to find a programming language that can be used in both scenarios. Because the validation in the instantiated model is most likely an extension of the Messaging Engine the Python programming language seems to be the perfect fit for the implementation of the outlier detection methods. This is supported by the availability of an extensive Python library (pyOD [ZNL19]), which implements various outlier detection methods.

The decision to use Python for the implementation of the validation eliminated the consideration of implementing the analysis in the frontend. As we could not find any maintained implementations of Python in JavaScript, one of the unmaintained implementations is PyPy.js [PyP15; PyP19] which only seems to support the deprecated Python 2.7.
Another option, which is also mentioned by the authors of the PyPy.js library, could be the WebAssembly based pyodide library [pyo23]. Pyodide, according to the authors, also supports libraries that use native C implementations, such as the NumPy library, an elementary component of the pyOD library [Zha23]. Making pyodide the best suited option to run validation in the frontend. Even with this option, there are some factors that point against executing the validation in the frontend, especially the potential performance problems, the lack of similarity to the execution of the instantiated validation and the associated additional effort to implement the training procedure.

To run the validation in a backend it had to be investigated, how the backend for the wizard will be built. Either as an extension of the existing Kotlin based Spring Boot backend used by the IoT Application Modelling Tool to store the models in a MongoDB instance, or as a stand-alone backend, completely written in Python.
The implementation as a stand-alone application makes the invocation of validation methods very simple, as the only step mandatory to do so is a method call. However choosing Python requires the reimplementation of components that are already provided by the Spring Boot backend, like the interaction with the database. This means that it is mandatory to establish a means to store the job data, for example by connecting to the same MongoDB instance that is used by the IAMT backend. The decision has to be made between a more complex handling of analysis invocation using child processes, when embedding the wizard backend into the IAMT backend, or by having to rebuild components that have already been configured in the IAMT backend, potentially introducing redundancies. Another argument against the full implementation in Python is the personal lack of familiarity when working with databases and HTTP APIs in the language, potentially increasing the time needed to complete the implementation.
Therefore, the implementation of the wizards backend was done as a component of the IAMT Backend, the Python implementations of the outlier detection methods are invoked using child processes.

After sorting out these fundamentals on how the backend should be implemented, more concrete aspects like the data model and the API specification had to be created.

5.3.1 Defining the Data Model

The analysis requires a mean to store unstructured data, such as images used for visualizations, text files like logs and `csv` files used to store the input and result data. For that a document called `Blob` is included. This document is used to store unstructured artefacts, either needed for the analysis or produced by it. These artefacts are identified by a universally unique identifier (UUID) [LSM05] like `5d27efe8-3eb0-4d6e-be63-a2ce2553675b` and the filename that corresponds to the type of file, for example, an input `csv` file may be stored as `5d27efe8-3eb0-4d6e-be63-a2ce2553675b.csv`.

Next, a means to link the input file with the node of the model is needed. This entity is called a `Job` and represents one usage of the wizard. It links the input file, which is stored as a `Blob` document and the node, stored by combining the identifier of the model and the identifier of the node, since only the combination will point at a specific node. Such a Job also gets a unique identifier, as a UUID, to allow one Node to have multiple jobs, for example, if an analysis may be rerun with a different dataset or modified input parameters.
The Job entity also stores all global parameters of the Job, including the mode of operation and the percentage used for training data.

A job is subdivided into tasks: for each of the implemented outlier detection methods, the source of which we will discuss in the next paragraph, a task is created as a child of the job. The output artefacts, such as the visualisation and the log files, are stored in a task, as are the measured results, such as the accuracy, the F_1-score and the runtime of the task. Due to the fact that the task is highly coupled to the job, the tasks are stored in an array of tasks within the Job object. In the database they are stored as one document. Implementing `Job` and `Task` this tightly coupled, had some unexpected consequences, we will discuss later.

The Outlier Detection Methods (ODMethods) are stored statically, in a JSON file, within the Java applications `jar` file. The ODMethod contains an identifier that must be unique, within the list of all implemented outlier detection methods, since it is used to identify the specific outlier detection method. In order to show the user a more in depth description of the outlier detection method, further attributes have been added storing the display name, a description to give a brief overview as well as links for further reading about the outlier detection method. To find the scripts associated with the outlier detection method the paths to them, within the `jar` file, are also stored in this definition file, by defining the entry point file, which is called to start the analysis as well as its dependencies. These files are also stored in the `jar` file, the root directory (`/`) therefore represents the root of the `jar`-archive. How one outlier detection method is defined in the JSON file is illustrated in Listing 5.1. The JSON file just contains an array of objects, following the same scheme as the one shown in the listing.

Both Jobs and Tasks have a state, that describes their current state of execution, the following possible states exist:

- **Pending**: The execution of the job or task has not started yet.
- **Running**: The execution of this job or task is currently in progress.
- **Done**: This job or task has been executed without any errors.
- **Failed**: Here we have to differentiate between jobs and tasks, since the state has different meaning for each of these entity types. A tasks state is set to failed, if the execution of the task has failed with an exception, whereas the state of a Job is set to failed if there is at least

Listing 5.1 Simplified JSON object defining the k-Nearest Neighbour algorithm

```
{
  "identifier": "knn",
  "name": "k-nearest Neighbor",
  "description": "REMOVED for readabliity",
  "furtherInformation": [
    "https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm"
  ],
  "isSupervised": false,
  "scriptLocation": "/od-wizard/python/knn.py",
  "dependencies": [
    "/od-wizard/python/base.py",
    "/od-wizard/python/benchmark.py"
  ]
}
```

one task that has turned into the failed state. If the job consisted of multiple tasks and one of them has failed the state of the job will become failed too. However, there will still be a usable result, assuming all other analysis tasks did not fail.

- **Cancelled**: This state is mainly relevant for jobs, it is meant for jobs that have been terminated upon user request, for example, by calling a cancel endpoint.

Figure 5.4 illustrates the core of the data model of the IAMT Backend including the wizard itself, it shows the relationships described above. However, some classes have been excluded to improve readability. The two excluded classes `TrainingResult` and `BenchmarkResult` are used to store the metrics, for both the analysis and the benchmark respectively. For benchmarks this includes the number of times it has been executed, the total time to perform the predictions, the average time for a prediction and the time needed for initialization. The analysis result contains much more information, such as the sized of the testing and training datasets, the convolutional matrices and binary classification metrics, including the accuracy and the F_1-score for the whole dataset, as well as for the testing and training portion. Additionally, it stores the time needed to perform the whole operation.

5.3.2 Outlining the API

Before creating API mappings, an investigation on what actions are mandatory to build the UI, as described in the wireframes, has taken place. As a result, the following operations turned out to be mandatory:

1. Upload / Retrieve unstructured data (Blobs)
2. Retrieving a list of the supported outlier detection methods, including their display names and description.
3. Creating a job
4. Retrieving a job object, including its current status

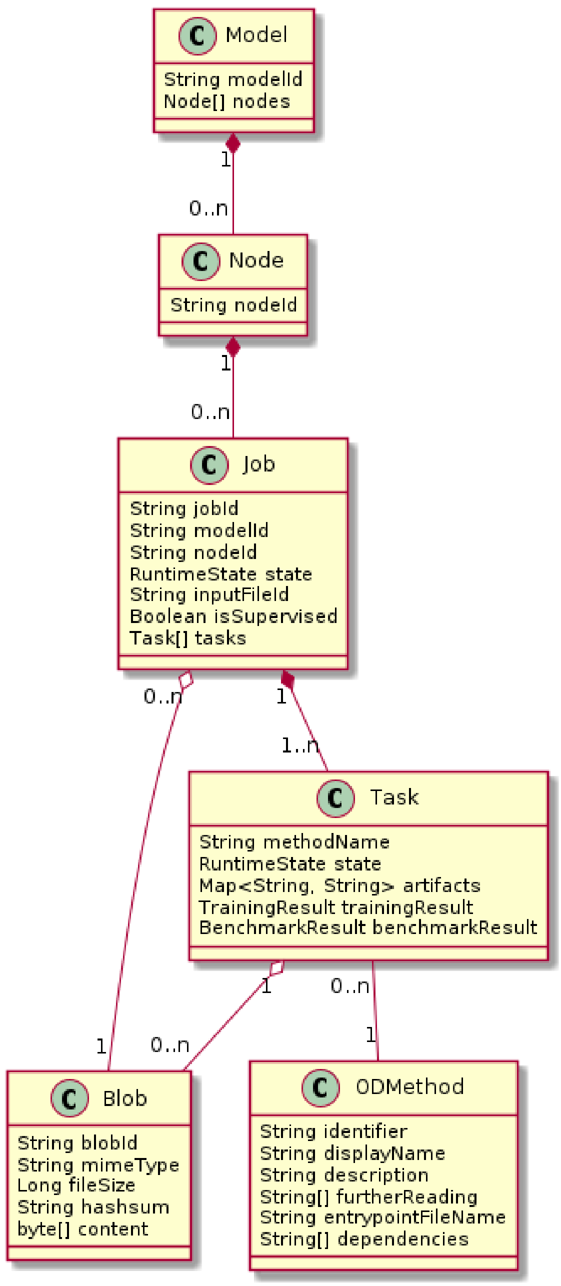


Figure 5.4: Simplified data model of the backend for the outlier detection wizard

5. Cancelling a job or task
6. Listing jobs based on several attributes e.g., for a node of a model or based on their state
7. Deleting a job

The API can be grouped into three sub-components, in the terminology of the Spring framework often referred to as Controllers [VMW23]:

- A blob controller, handling the upload, retrieval and deletion of blobs,
- a controller that provides the list of outlier detection methods, either as list or just the single element, referred to as the method controller and
- the main controller, responsible for handling all interactions with jobs, such as creation, status retrieval and cancellation.

Most of these operations, apart from creating or cancelling a job, are simple CRUD (Create, Read, Update, Delete) operations, that check the input for validity and then either store, retrieve or delete the data in the database. As shown in the list of operations above, there are no real update operations, apart from the job cancellation, which has to be handled differently. Because, it has to terminate running processes if the job is in progress. How cancellation has been implemented will be discussed in **??**. Because these implementations did not come with any technical challenges or questions regarding the system architecture, they will not be discussed further.

Apart from Job creation and cancellation, there is another API function, that may need more complex implementation: How will the frontend get notified about job status changes if it is in the *Processing* step of the wizard?
In order to make a reasonable decision, a quick investigation on how the frontend should update the status of the job had to be done. There were two options to retrieve the status of a job, either by using a push or a pull approach. In the push approach, the WebSocket technology, supported by almost all modern browsers would be used, as it allows the server to send data to a client, even without a prior request. Of course, it is still mandatory to open the WebSocket connection, to clarify who is listening for messages [Moz23].
The other suitable option is pull-based. Here, the frontend application queries the server in a recurring interval e.g., every five seconds, to determine whether or not the status has changed. Both of these approaches have pros and cons. However, the decision on what to choose is also dependant on the use-case.

Some pros (+) and cons (-) of the WebSocket based solution include:

+ Provides almost instant status updates, making sure the frontend is never outdated. The only factor limiting this is the network connection, which may introduce latencies.
+ Less susceptible to high server load or even overload when many users use the application at the same time, since the backend will notify connected frontend applications once a change for their job has occurred.
- Implementing the WebSocket based mechanism is relatively complex, as new controller mappings and means of notification, within the backend have to be introduced.

- May cause issues in combination with the IAMT-Gateway, requiring adjustments in it, to also forward WebSocket connections

The polling approach, on the other hand, is much easier to implement, as is does not need any further additions to the backend apart from the *Retrieving a job object, including its current status* mapping. Because, our design intends this mapping to also be used to build the result step of the wizard, this endpoint would also be needed for the WebSocket based implementation, reducing the implementation effort of the poll based approach in the backend to zero.
This tool is intended to be used by a small user base at the same time, which reduces the risk for high server load significantly. In combination with a decent time interval, the increased server load, does not represent a huge issue for our use-case. To give a small example, we assume there are 10 users, a relatively high number for one instance the IoT Application Modelling Tool, each submitting a job at roughly the same time. With an update interval of five seconds, this results in $10 * \frac{60}{5} = 120$ requests per minute, a number that should not cause any issues on most host systems. Another option to mitigate the risk of high server load, is increasing the wait duration between requests. Doubling the interval to ten seconds will halve the number of requests per minute, with the drawback that the user may get the information that the job has been completed after a short delay. However, the analysis process can be considered a comparably long running task that at least takes several minutes to finish. With this in mind the potential delay of ten, or even five, seconds to get the status update, becomes negligible.
For these reasons the decision was made on the polling based status update approach.

5.3.3 Handling the execution of jobs

Jobs are considered long running tasks, meaning that they will not be executed directly upon submission. Instead, they are executed asynchronously, in a separate thread. How this is handled and what kind of restrictions this introduces will be discussed in this section.

The reasons behind this asynchronous approach is simple: a job may need several minutes, if not hours to complete its analysis. Making synchronous execution, i.e., running the job within the same thread in which the HTTP request is also processed impractical, because the HTTP request sent by a client requesting job execution has probably already timed out at this point. For this reason, we had to implement a processing approach that is asynchronous, implementing long running jobs. Implementing this concept is usually done in three components, a producer, a queue and one or more consumers. The producer in our case is the user, at least indirectly, since the user initially triggers the creation of a Job using the API that will in turn create a job object, stored in the database and gets enqueued for further processing in the queue component. From there, a consumer, i.e., the job runner component, takes the enqueued object and starts processing it asynchronously [Sha21].

From a design perspective, the decomposition of these three components into separate small applications, could be considered. In this case, the consumer and the producer would have to be implemented and the queue could be handled using an off the shelf message-broker software, like RabbitMQ[11]. If the application should be usable for a large user base, with the capability of scaling out the job execution dynamically, this much more complex implementation approach could be considered. However, the tool is more likely to be used in many small instances used by a small user base ($\leq$ 10 Users). On that scale, building the application in such a highly scalable, cloud-native, fashion seems to be overcomplicating things significantly.

For this reason, the basic idea of producers, consumers and a queue is implemented within the application using the Java standard API, Instead of using a queue in the fashion described above, we use a `ThreadPool` that is used to schedule the execution of jobs. This `ThreadPool` represents the queue and parts of the consumer, since it keeps a queue of Java `Callable` objects, that in turn implement the execution of jobs and their corresponding tasks. The pool then consumes as many `Callable` objects from its queue as it has threads available. For example, a single threaded `ThreadPool` processes only one job at a time. If the pool has four available threads it can process four `Callable` objects at the same time. The producers, in the terminology of Java also referred to as submitters, they submit an instantiated `Callable` object to the `ThreadPool` in order to schedule it [Eug22; Ora14].

The execution of a job works as follows:

1. The user invokes the *Create Job* API endpoint.
2. The backend creates a new `Job` object, stores it in the database, creates an instance of the `Callable` responsible for the processing of the job and submits this `Callable` to the `ThreadPool`, queueing it for execution.
3. Once compute resources are available, the `ThreadPool` schedules the job, to be more specific the `Callable` responsible for the processing of the job, on one of its available threads.
4. The `Callable` performs the analysis and stores the result.

The Job execution is implemented in such a way that the current state is always persisted in the database, enabling the ability to always get the current progress using the *Get Status* API Endpoint, ensuring the frontend can be updated with the current state by using polling, as discussed before. The job execution iterates over the tasks of the jobs, runs them, in a manner we will discuss in Section 5.3.4, and retrieves the artefacts produced by the subprocess. Depending on the result of the task, its state will be set accordingly. Once all tasks have been executed the state is either set to `Done`, if all tasks have ended without any issues, or `Failed`, if at least one of the tasks did not terminate successfully.

Next, the task execution procedure and the implementation of outlier detection methods will be discussed.

5.3.4 Handling python subprocesses

In order to use the implementations provided by the pyOD library, to perform the outlier detection and our previous decision to extend the already existing backend of the IoT Application Modelling Tool, implementing a mechanism that allows us to run both Python code and Kotlin code, to perform the job execution was necessary. The solution for this problem are child processes of the Python programs that get invoked using the `ProcessBuilder`, which is part of the Java standard library [Jon19; Ora14].

[11] https://www.rabbitmq.com/

Listing 5.2 Reduced output of the `env` command on a Linux machine before activating the virtual environment

```
PWD=/home/chris/venv-demo
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/snap/bin
```

A task will execute two child processes in a row, the first one is responsible for performing the main analysis, i.e., training the model and performing the predictions on the test portion of the data set, in order to compute the metrics for binary classification, if possible, like the F-score and the accuracy, and draw the visualizations. The second one performs a prediction benchmark, measuring the runtime of the outlier detection method for loading the trained model from the previous task and running the prediction on a predefined number of rows. In order, to get an average processing speed per row.

With this very rough idea how the implementation will be handled, there are still many open questions that needed further clarification in order to implement the execution.

5.3.4.1 How do we provide a runtime environment for Python scripts?

Since Python is a requirement for the execution of Jobs, it is assumed that it is already installed. But even with Python available there are still many dependencies of the pyOD library that have to be installed and provided somehow. The virtual environment module which is part of Python itself is the perfect solution for this. Since it allows the creation of an environment that is specific to one application. Instead of installing all dependencies globally, which would be the other option to provide the dependencies [Pyt23].

Which option should be chosen has to be decided by the person that is in charge of deploying the application in the end. This is not a decision that can be made in general, it has to be decided in a case to case basis. For example, if the IAMT Backend application is running on a developer machine, without Docker, the use of a virtual environment makes a lot of sense. In order, to isolate the dependencies from other project on which the person may work. If the application on the other hand is deployed on a virtual machine, or within a Docker container, with the sole purpose of running the IAMT backend installing the dependencies globally does not cause any issues. The application must therefore support both virtual environments and globally installed dependencies. To understand how this could be implemented in the application, it is mandatory to understand how virtual environments work.

To use a virtual environment in Python, it has to be activated by running an activation script. As illustrated in the simplified outputs of the `env` Linux command, which prints the current environment variables to standard output, shown in Listing 5.2 and Listing 5.3, this activation script alters the `PATH` variable and adds other environment variables that point to the location of the virtual environment [Pyt23].

With the functionality of virtual environments figured out, it was mandatory to find out how the virtual environment could be used within a process that is invoked by the Java `ProcessBuilder`. The answer to this problem is very simple: it is just necessary to ensure the environment variables

Listing 5.3 Reduced output of the `env` command on a Linux machine after activating a virtual environment called `env`

```
PWD=/home/chris/venv-demo
VIRTUAL_ENV=/home/chris/venv-demo/env
VIRTUAL_ENV_PROMPT=(env)
_OLD_FISH_PROMPT_OVERRIDE=/home/chris/venv-demo/env
_OLD_VIRTUAL_PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/snap/bin
PATH=/home/chris/venv-demo/env/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/
bin:/snap/bin
```

with which the Java application have been launched are passed through to the child-processes. The default implementation of the `ProcessBuilder` already handles environment variables this way by default, making no implementation changes necessary [Ora14].

To use a virtual environment with the application it is therefore only mandatory to start the Java application from within a shell-session that has the corresponding virtual environment activated.

The capability of running the script is only one portion of having a suitable runtime environment. It is also necessary to make sure every task has a working directory, it can perform its actions in. To avoid any conflicts, for example, caused by identical file names, it should also be made sure the directory is unique.
To implement those unique working directories, it is initially necessary to create a directory in the systems temporary directory or a location chosen by changing a configuration option. Within this directory a child directory for every Job is created. Which, in turn, will contain a child directory for every task of the job. An example of the format used to generate these unique working directories is `/tmp/iamt-od-wizard/<JobID>/<TaskName>`. This directory gets created for every task that gets processed, after the task is terminated and the artefacts have been saved the temporary working directory is removed, to save space.

With the working directory clarified, there is one more important thing that has to be done to build a suitable runtime environment: providing the script files. This is very basic, as discussed before the script files are contained within the Java resource folder, that becomes part of the `.jar` file during the build process using Maven. The script files needed by each outlier detection method are declared in the definition file.Listing 5.1 is an example, showing a snippet that is used to define the k-nearest neighbour algorithm. To build the runtime environment, when a task is getting executed, the backend copies the files defined in the `dependencies` and `scriptLocation` to the previously created working directory, using the Java `ClassLoaders` `getResourceAsStream(String path)` method. In order, to retrieve the content from the `resources` folder or `.jar` file.

After copying it is not necessary to set the executable flag on these files using the `chmod +x <Filename>` command since these scripts can also be invoked using the `python3 <Filename>` command instead. Invoking a program using this command tells the python program to read and execute the file, while the process is much more complicated if the script should be made executable. In this case, the entry point files must contain a shebang, e.g. `#!/bin/bash` line, that tells the operating system, which program should be used to run the code that follows [FOL23; Mas01]. Apart from the requirement that a file must start with a shebang, the use of this mechanism comes with dome more drawbacks that can be avoided, by invoking the Python runtime directly. These include, the requirement that files must be made executable before invoking them and the potential platform incompatibility, as the shebang is only generally supported by UNIX-like operating systems. On Windows, support can be added by using tools like Cygwin[12], introducing further, unnecessary, dependencies.

5.3.4.2 How are parameters and input data passed to the Python scripts?

After the runtime environment has been created, the task is almost ready to be launched, only the input parameters have to be passed into the script and the input dataset has to be stored in the working directory for processing.

Providing the input dataset file is very simple, here we just copy the contents of the input dataset from the database into the working directory, with a common filename, like `input.csv`, that the analysis and benchmark script will then read as basis to get data.

The more sophisticated question is how input parameters, such as the testing percentage, or the flag that the is set if the input file should be randomized before performing the analysis, are passed into the scripts. There are many options to do this, such as creating a configuration file, with a unique name or using command line arguments to get these parameters into the application. However the simplest option that is also very failure proof is the use of environment variables to do this. Using environments variables does not come with any requirements in regards to libraries, as the use of custom environment variables is supported by both the `ProcessBuilder` and the Python standard library. The only requirement to ensure environment variables are working properly is the declaration of names. The names of environment variables must be identical in both the Python script and the backend implementation. This introduces redundancies in the code, that may eventually cause problems if the names are only altered in one part of the application, during a refactoring of the source code at a later point in time. There are solutions to resolve this, such as using a shared configuration file. However, implementing such a mechanism was decided against because this seemed to make the system overcomplicated, as most developers are often aware of those kinds of problems.

Using environment variables also allows the setting of default values, that will be used if the environment variables are not set, easing the development of the Python scripts, as they can be invoked directly, without tediously setting the environment variables before invoking the script.

5.3.4.3 How is the result of the process returned to the parent process?

We previously discussed how parameters are passed into the python application. To make sure the two processes can interact it is also mandatory that parameters can be retrieved from the Python child process. For parameter output there are just as many options to handle them as there were with the input handling. The traditional error code is not sufficient, for the type of interaction that is needed in our use case. Since the exit code is only one integer value, not sufficient for the amount of data that has to be transferred back to the parent process. The potentially most feasible options are: (1) an output file, with a predefined name, that contains a serialized object with the return

[12]https://cygwin.com/

values, for example in JSON format, or (2) passing the return values to the parent program using the standard output or standard error streams, for example by writing a non-human-readable JSON object, i.e., a JSON object printed without any line breaks, between key-value pairs. This makes it possible to look for JSON objects by performing a line by line analysis of the standard output or standard error streams, looking for opening brackets at the start of the line and closing brackets at the end of the line.

Because both standard output and standard output have to get processed anyway, to store their outputs as a logfile, that can be used for debugging, in case something goes wrong, the second option is chosen. To be more precise the Kotlin backend expects a JSON object in one line of the standard output. The backend searches for such an object, if none is present an error message is thrown. Since that is a clear indication, that something has gone wrong.

The object that is written to standard output is expected to follow a predefined schema. In order to be parsed by the `ObjectMapper`, provided by the Spring framework, to convert it to a Java object. For the training part of the task, this object also contains a list of the files the script has produced. After termination, the backend copies these files into the database and stores their corresponding identifiers within the object instead of the original filenames, to keep the files beyond the duration of the task, after the working directory has been cleaned up. The files are mapped by key and value, the key is used to identify the purpose of the file, of which the path is set in the value, an example for the key is `visual` which represents the visualization.

5.3.4.4 What is implemented in the training script?

As briefly discussed there are two scripts for every task, the training and the benchmark script. The training script is performing the initial analysis, which will result in visualizations and the binary classification metrics. To achieve this the training scripts performs the following steps, it is important to note that some of these steps may only be executed if one or more preconditions are fulfilled.

1. In the initial `Data Loading` step the, script reads the dataset, randomizes it with a preset seed, if the user enabled this feature for the job and splits the dataset into training and testing portions based on the split percentage provided.
2. After loading the data, the model is trained, by invoking the train method in combination with the training dataset. The training also produces predictions for the training dataset. These predictions are only useful for further analysis if the outlier detection method is unsupervised. In that case, these predictions illustrate what the outlier detection method thinks are outliers in the training dataset. In case of a supervised outlier detection method, these predictions are of no value, as the model has already been taught what the value should be in those cases.
3. At this point the prediction can take place, by performing a prediction on the testing dataset.
4. If the input data is labelled, so that rows are declared as an outlier or an inlier, the expected value for the row is available, the numbers of false and true inliers and outliers can be collected in order to build a convolution matrix for the testing dataset. We also compute the latter for the training dataset, however the result of this is only providing useful information if the

implemented outlier detection method is based on unsupervised-learning. For completeness this process is also done for the complete dataset, the accuracy and *f*-score of this dataset, represent the weighted average of the two datasets.

5. If the outlier detection consists of either one or two dimensions, the visualization is drawn, visually illustrating which value is considered an outlier and which one is not. The types of visualizations has been discussed in Section 4.3.2, the visualizations have been implemented using the `pyplot`[13] library. To visualize two dimensional datasets, the implementation to perform the visualization provided by the pyOD library is used. For one dimensional datasets histograms with bins, if the value range exceeds 250, have been added. A visualization for three dimensional datasets is not provided due to potential lack of usefulness and the the significant challenges that would be faced when implementing it, as discussed in Section 4.3.2.3.
6. Finally, all result data files are stored within the working directory, including, the visualization, a list of mismatches, the predictions the program has made, and the instance of the outlier detection algorithm implementation that is exported using the `joblib` library. As recommended by the developers of pyOD to make trained models persistent [Job23; Zha23].

The training script, for an outlier detection method, consists of two files, a file specific to the outlier detection method and another, generic file, that contains the generalized implementation of the whole procedure. In the method specific implementation, it is only mandatory to instantiate the outlier detection method, provided by the pyOD library, and to call the generalized method to perform the analysis. This is possible, because all implementations of outlier detection methods provided by the pyOD library are derived from the same base class.

5.3.4.5 What is implemented in the benchmark script?

The process of benchmarking is only implemented in a generic manner, since the previously stored `joblib` object is loaded, this instantiates the outlier detection method in an instance that is identical to the object created during the training process. Doing this also makes sure, the created `joblib` file can be used to reopen the model, a task that becomes very important if the outlier detection method has to be instantiated on an IoT Device.

The benchmarking process loads the model and the initial dataset from the working directory. After loading, a random sample of 100 rows is selected from the original input dataset. At this point, a time measurement is started and then every one of the rows is predicted by invoking the predict method for every one of these rows. After the predictions are done, the time is stopped as a result the average computation time for 100 rows can be calculated. This is the main metric derived from the benchmark.
The result is then returned in the same manner as the training process.

In case of failure of the benchmark process the execution of the task is not marked as failed, it is instead marked as succeeded, however the benchmarking results, which do not exist, will not be displayed in the frontend. Instead a warning that the benchmark has failed is shown.

[13]https://matplotlib.org/stable/tutorials/introductory/pyplot.html

Listing 5.4 Outlier detection method specific implementation for python of the *k*-nearest Neighbour algorithm

```python
# Import the implementation of the outlier detection method
from pyod.models.knn import KNN
# Import the generic implementation
import base

# Instantiate the outlier detection method
inst = KNN()
# Run the training procedure
base.run_od_method(inst, "knn", False, base.DEFAULT_FILE_NAME)
```

5.3.4.6 How can new Outlier Detection methods get added?

The addition of outlier methods is simple, assuming the method is implement by the pyOD library, or the implementation is derived from the same base class as the implementations part of the pyOD library. The addition is done in two steps: (1) The creation of a method specific script file and (2) the addition of the method to the list of implemented methods.

The method specific implementation is very simple, it is only mandatory to import the common library and the implementation of the outlier detection method. Then the implementation must be instantiated and the analysis must be launched. By calling a method from the common file called `base` as illustrated in Listing 5.4. Calling the `base.run_od_method` requires some parameters, including the instance of the outlier detection method, the name used to identify the outlier detection method, this name, preferably, is identical to the unique identifier defined in the list of the outlier detection methods. The third parameter is a boolean that is set to true if the implemented outlier detection needs the result for training, i.e., this value must be true if a supervised outlier detection method is implemented. The last parameter defines the filename of the initial dataset, which should always be identical, as it is stored in a constant.

The base script also needs more parameters, for example whether or not the input dataset is labelled, referred to as *Has y-data*, those parameters are not constant and depend on the current task they should process. For this reason, these values are passed through the script using environment variables, as discussed before, the implementation of this is completely handled within the base script itself.

In the second step, the method must be added to the file defining all implemented outlier detection methods. Here, an object like the one illustrated in Listing 5.1 has to be added. To support a new method the values must be changed accordingly, the script location should point to the script created in the previous step within the resources folder of the applications source code. The dependencies must include the `base.py` and `benchmark.py`, as they are used for the benchmark and as a library, used both during the benchmark and training.

5.3.5 Handling job cancellation

Job cancellation can be divided into three types of cancellation: (1) The cancellation of a job that is still in the `Pending` state, (2) the cancellation of a running job, that is already performing the analysis or is running the benchmark and (3) the termination of the currently running task, to break out of a dead locked process without completely terminating the Of course the cancellation of jobs that already completed the execution, either successful, unsuccessful or upon cancellation cannot be cancelled. As they have already terminated. The latter two of these options are the challenging ones, as there are child processes that have to be terminated, too. Before looking into those types of cancellation, the cancellation of pending jobs will be discussed.

Cancelling a non-running job is rather simple, only the removal from the list of scheduled jobs, and the modification of the jobs state to `Cancelled` have to take place. Due to the asynchronous implementation and the lack of atomicity in the cancel operation it is theoretically possible, that the execution of a job starts at the exact same time as the job gets cancelled, which may result in a delayed termination of the job, i.e., the first task may be launched. But after its termination the job will also cancel itself, since it is marked for cancellation and this marking is checked every time a new task should be started. A lock or mutual exclusion (mutex) can resolve this problem. However, the scenario is very unlikely and even if it occurs the job will still get cancelled with a bit of a time delay. For that reason, the more complex locking mechanism was not implemented.

The termination of running jobs and also just terminating the currently running task is more complicated. Since the cancel request is sent asynchronously, i.e., in a different thread than the one orchestrating the job execution. The main problem with this type of cancellation are the child processes that are created by the callable responsible for the job execution. As it is waiting on the completion of one of these child processes most of the time. In case a job gets cancelled, these child processes have to be terminated, before the job can be marked as cancelled, to make sure the job document does not get turned into an illegal state, caused by a lost update scenario.

For the implementation of this termination mechanism multiple options have been investigated:

- In a UNIX-based environment, there is always a `kill` command that can be used to terminate a process, assuming one is permitted to do so. This is a very primitive version of the implementation, that was initially used to terminate some tasks during development, for example because an outlier detection implementation has deadlocked. Here manually invoking the `kill` command with the corresponding process id (`pid`) which gets logged by the backend, in a Terminal, allowed the quick and easy termination of tasks. Since termination using `kill` usually results in a non-zero return code of the child process, the callable thinks the process has failed, marking the corresponding task as failed, however the rest of the job will continue to execute just like before.
  With these observations, a potential implementation of the cancel approach can be derived. For this the process ids (`pid`) of the currently running child-process have to be stored in a thread-safe Map that maps the job id to the associated process ids. Every time a new child-process gets invoked the resulting `pid` must be stored in this map to allow the termination from a separate thread. If the user now invokes the cancel endpoint another child-process of the `kill` command will be launched, terminating the child process, currently running in the callable.
  This procedure to this point, only terminates the currently running task, while this is one of the types of cancellations that is offered, the cancellations of whole jobs is not possible

yet. To achieve that, further modifications have to take place. In order to avoid lost updates, the job should be marked as cancelled within the running callable, that will then terminate upon marking the job and the remaining tasks as cancelled. To make that possible another thread-safe collection, specifically a set, has to be created, If the user requests the termination of a running job this job id will be added to this set, and afterwards the child-process will be terminated, just as discussed previously. Once the job executable wants to run the next task it ensures the job it is currently processing is not in this set, if it is present it will not run the next task, instead it marks the job and all pending tasks as cancelled before terminating [Lin22; Neg20].

- The previously described approach, relies on system binaries to perform the termination, this may cause issues, based on the operating systems, another, more platform independent option is the use of Java API to terminate the process. One solution in this case could be: the use of the Callables `cancel()` method, which will in turn cause the corresponding thread to interrupt when waiting for the child-process to terminate, by catching the `InterruptException` while waiting for the process to complete. In case this exception gets thrown, the process could then be terminated using the `destroy()` or `destroyForcibly()` methods. After terminating the child process, the callable must mark the job or task as cancelled respectively [Ora14].

At the current stage of the implementation, the backend uses the first approach to cancel jobs, the main reason is the reduced implementation effort, coming from the initial non-cancellable implementation.

As briefly mentioned, there are two different types of cancellation, the termination of the whole job and the termination of just the one, currently running task. The task termination option is intended to be an emergency switch to kill a child-process that has locked up or just takes too long. Because the dataset is too large. Unlike the cancellation of jobs, the termination of tasks is only possible if a child-process is running, otherwise this action will not do anything.

5.3.6 Handling illegal application states and outdated datasets

Our concept comes with several caveats, caused by the loose enforcement of relationships and the implementation of the job scheduling mechanism. In order to mitigate these problems, some measures have been implemented to perform a clean-up on the database and to handle jobs that were running, when the application was terminated.

5.3.6.1 Database clean-up

The relationships in our data model are not enforced by constraints within the database. The reason for this is the intended looseness of the relations. For example: a `csv` Blob may not be associated with any Job or Task, if someone wants to reuse the data without uploading it again[14].

Since those loose relations may result in orphan objects, such as jobs or blobs a clean-up mechanism has been implemented to get rid of these unused elements. The clean-up service is implemented as a cron job that is executed every 12 hours. Orphan documents also still have a timeout, to make sure they are kept at least seven days before they get removed. This is particularly important to ensure that a blob could not possibly still be used in a job that has not yet been created.

These clean-up mechanisms can be disabled by a config option, in the `applications.properties` file or by any other mean supported by the Spring Framework to alter these values, such as command line arguments [VMW23].

5.3.6.2 Job cancellation on startup

The implemented mechanism of execution is not failure tolerant, which in consequence could cause jobs to be running, even tough they are not running any more. This occurs if there are jobs in execution, when the main application crashes or gets terminated. At that point, the jobs are stored with the state `Running` in the database, when the application is restarted they would stay in this state forever (unless manually cancelled), because there is no mechanism to reschedule the jobs after the application crashed.
To avoid this, a method that is invoked, once every time the application launches, marking all jobs that are stored as running in the database as cancelled.

Another option could be the rescheduling of the jobs with `Running` state on startup, however due to the possibility of the still running task being the cause for the crash, for example by causing an `OutOfMemoryError`, we decided against this option to prevent potential cash loops that could only get resolved by manual intervention.

5.3.7 Making the Backend portable

Since the IAMT backend has a lot of runtime dependencies, like libraries, after the wizard has been implemented, a means to make the distribution easy, especially to make the process of deployment as simple and deterministic as possible during the evaluation, has to be implemented. To fulfil this the backend has been dockerized.

In the previous implementation of the Dockerfile the application was built outside of Docker and the produced artefacts (`jar` files) were copied in the Docker image. Depending on the complexity of the build process this may be a good option, however the build process of the application is not complex, i.e., it does not consist of many child projects and closed source dependencies that may need special credentials to build. The project can be built by running `mvn clean package`, making the compilation process very simple. For this reason, the benefits of building the Docker image over weigh the downsides, as the building of the application using `docker build` always creates an identical base environment, which in turn ensures a consistent build environment. Building the application for the Docker image is therefore done in the first stage of the Dockerfile, after the application is built the second stage is initiated, here all dependencies fetchable from the package manager, like Java and Python are installed. After the installation of the base dependencies the python dependencies like pyOD, Tensorflow and xgboost are installed. To reduce the size of the output image the cached package files get removed, all of the previous steps are executed in one `RUN` statement, to make sure all the dependencies are added in one layer of the docker image, this

[14]The frontend design does not have this feature in mind, however it may be implemented later or the backend application may be used with another frontend supporting this.

Listing 5.5 Commented dockerfile of the backend

```
FROM eclipse-temurin:11 AS builder
WORKDIR /iamt-backend-build
# Copy sources
COPY . .
# Build application, excluding any tests
RUN ./mvnw package -DskipTests

# Output Image
FROM ubuntu:latest
# Ensure install passes, without failures due to missing input from the user
ARG DEBIAN_FRONTEND=noninteractive
# Install Dependencies and remove cached files
RUN apt-get update && \
    apt-get install -y openjdk-11-jre-headless python3.10 python3-pip && \
    python3 -m pip install pyod tensorflow xgboost && \
    rm -rf ~/.cache/pip/* && \
    apt-get clean
# Expose application port
EXPOSE 8080
# Set the copy target directory
WORKDIR /iamt-backend
# Copy artifact from build stage
COPY --from=builder /iamt-backend-build/target/*.jar .
# Set directory for job workdirs
WORKDIR /iamt-workdir
# Set Entrypoint (Simplified)
CMD ["java", "-jar", "/iamt-backend/iamt-backend-0.0.1-SNAPSHOT.jar"]
```

also has a great effect on the size of the output image. The build is finalized by getting the binary built in the previous stage and by setting the entry point, for launching the container later [Bib22; Doc23].

A commented version of the dockerfile is depicted in Listing 5.5. Building a Docker image always needs some kind of base image. For the backend image, `ubuntu`[15] was chosen as the base image, as it supports the use of the `apt` package manager and the names of the packages are identical to the ones in the Ubuntu operating system, making the setup of the image very simple and understandable, especially for someone who has used Ubuntu beforehand.
The compilation of the Java binary is done by using a Temurin[16] base image. Termurin is one of the many available OpenJDK runtimes that have been derived since Oracle has announced changes to the license of the original Java Product. This image also includes Maven[17], which is used for dependency management and compilation of the binary.

[15] https://hub.docker.com/_/ubuntu/
[16] https://adoptium.net/temurin/
[17] https://maven.apache.org/

5.4 Frontend Implementation

The backend was developed to suit the needs of the wizard, that has been outlined in the previous chapter, with the implementation of the backend, a basis has been made, that can be used to implement the outlier detection wizard in the IoT Application Modelling Tool, derived from the previously discussed concept. This section will discuss the implementation of the concept, challenges faced during this process and things that have been altered or added in comparison to the original wizard.

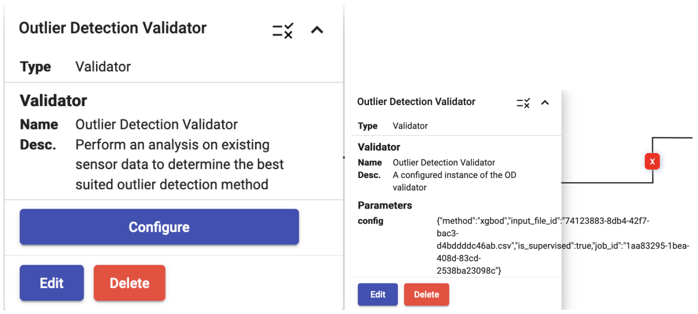
5.4.1 Adding instantiatable nodes

In order to get validators that can be configured after they have been dragged into the modelling space, a new type of node, currently limited to validators, had to be introduced: the instantiable validator. Since, the validators introduced in Section 4.4 are not sufficient for this purpose, as they only can be configured by setting configuration parameters, as we saw in Section 4.4.2.

To make this possible, we have to introduce a placeholder type, that allows the modeller to completely model the data flow. After the modelling of the data flow is completed the basis for assisting is available, as we have described in Chapter 4.

Such an instantiable node consists of two node types: 1. The uninstantiated type, that shows a configure button, is listed in the selection menu on the left and has no input parameters, as well as 2. the instantiated type, which is not shown in the selection menu, and also has a single, read-only unmodifiable string parameter called `config`. How this value is set will be discussed next.

To instantiate such a node, the assumption, that there will be some sort of assisting UI guiding the modeller through the instantiation of the node, such as a wizard. After the completion of this program we expect a string return value from it. Based on the string returned, we either instantiate the node or the node stays as is, in case the UI has failed or has been closed. The returned string is intended to contain the configuration for the instantiated node. At first glance a single string may not be sufficient. However, the usage of serialized objects, such as a Base64 encoded string or a single line JSON, is intended to be used to store more complex configurations.


Outlier Detection Validator
Type Validator
Validator
Name Outlier Detection Validator
Desc. Perform an analysis on existing sensor data to determine the best suited outlier detection method

Outlier Detection Validator
Type Validator
Validator
Name Outlier Detection Validator
Desc. A configured instance of the OD validator
Parameters
config {"method":"xgbod","input_file_id":"74123883-8db4-42f7-bac3-d4bddddc46ab.csv","is_supervised":true,"job_id":"1aa83295-1bea-408d-83cd-2538ba23098c"}

Figure 5.5: An uninstantiated validator-node (left) in comparison to an instantiated validator node (right)

The process works in the following steps:

1. The user adds an instantiable node, from the selection menu to the model. After doing this instantiation (deployment) of the model is temporarily disabled.

2. The user models the connections to this node within the model.
3. After expanding the node, the user clicks on the configure button, that will open a wizard, or a similar UI, to start the instantiation of the node.
4. Once the steps have been completed or the process has been cancelled, the UI has to return a string value. If the value returned is not `null` or empty the type of the node is changed to its instantiated variant and the modeller is allowed to proceed with the instantiation of the model, assuming, that there is no further uninstantiated node in the model.

The visual difference between an uninstantiated node and its instantiated counterpart are illustrated in Figure 5.5.

It was also important to introduce a mean to enable the change back to the uninstantiated type, without having to replace the node. To make this possible the dialog to modify the validator, which is a simplified version of the dialog used to modify operators, seemed to be the simplest to implement option. Therefore, the dialog was expanded to support the selection of the uninstantiated type, while the instantiated type is selected. Since the user should not be able to alter the parameters of the instantiated validator, as they are set by the configure function. The mechanism that allows the modification of parameters had to be disabled, if the node is an instantiated validator. To prevent the instantiation with an empty parameter, the modeller can only change to the uninstantiated type. If the uninstantiated type has been selected, he cannot select the instantiated version again, without leaving the dialog, by closing or cancelling, and reopening it again. When the edit procedure is cancelled or closed, by clicking outside of the dialog, the type is not altered. The modeller must click the save button to perform the action.

5.4.2 Implementation of the wizard

The implementation of the wizard itself was done as, separate module, in Angular terminology, within the IoT Application Modelling tool. To keep the coupling between the wizard and the modelling application as loose as possible. Of course, the wizard is still tightly integrated and may not be used independently, without significant modifications. However, reducing the coupling still eases the process of reusing the components, separates concerns within the application and simplifies the source code, to generally follow best practices [Woz22] The wizard user interface is opened as a modal dialog, that is intended to be only opened once for every instance of the UI, at the same time. This is not a limitation, it is a design decision, since the modal dialog will be in front of any other component in the frontend application.

Before taking a look at how the steps have been implemented, some aspects in regards to the implementation behind the templates, i.e., what is displayed to the user, will be discussed.

One important task in such a wizard, or any more complex web application, is the management of the application state. This type of state must not be confused with the different steps in the wizard, since the flow between them can be modelled as a state machine, as we saw previously. Instead, the state should be considered something that is used to store information in addition to the wizard state, in state machine terms, like the node that has been selected or the mode of operation. This data has to be stored in a way, that makes is persistent within the application instance, like a singleton. To handle this a singleton service, called the state manger has been introduced, its responsibility is the centralized state management of the wizard. It also acts as a mechanism for component interaction across larger hierarchies. Because, this otherwise would have to be handled using inputs and outputs within the component tree. These interactions are handled by asynchronous `Subjects` on those a component can publish a change and another method can subscribe to it. One example of this are the three subjects that are used to enable/disable the Next, Back and Cancel buttons. Every time the state should change, here there is a subscription for every one of those subjects, waiting for change events. We mostly use a specific type of subject: the `BehaviorSubject` which allows the definition of an initial value and also stores the current value, also a new subscription is always invoked with the current value [Bri23; Dev23a; Nya20]. All values provided by the state manager are stored using `BehaviorSubjects`. These mechanisms are provided by the RxJS library[18].

One important question that had to be investigated, before the implementation of the steps started: How does the wizard dialog change steps? The Angular router could be used, that allows the modification of views, based on the current URL, however since the frontend application in its initial state does not use a lot of routing this option would probably require more effort, in comparison to the option that was chosen. Instead, the state manager stores the current step using a `BehaviorSubject`. The corresponding step component is shown if the current step matches the step identifier using the `ngIf` directive. This mechanism is sufficient for this use-case since there are only a small number of step types that are introduced.

The parent component of the main wizard view also contains three commonly used buttons: Next, Back and Cancel, that can be enabled using the state manager. The state manager also provides a mean to change the behaviour of the buttons based on the current step the wizard is in.

Another service introduced interacts with the Backend API, this service wraps the API methods in TypeScript methods, by calling the HTTP method using the HttpClient provided by Angular.

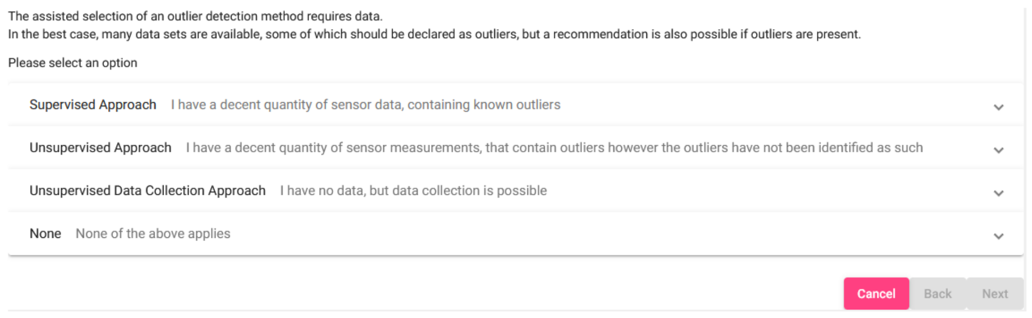
5.4.2.1 Step 1: Introduction

After the Wizard is opened for the first time, or when no job has been created previously, for the specific node, the introduction step is shown. Its main purpose, is the selection of the mode of operation, by informing the user what requirements have to be fulfilled to perform an analysis. This page also shows, which outlier detection methods are supported for the corresponding mode of operation, since the unsupervised mode is not able to perform an analysis with supervised machine-learning based outlier detection methods.

Each outlier detection method is described with a short description and some links for further reading, that for example point to the publications that have proposed the corresponding outlier detection method. These informations are shown using expandable panels, If the user wants to view these informations, the corresponding expansion panel must be opened, by clicking on the name. Afterwards, the panel expands and the informations related to the outlier detection method is displayed.

The selection of the mode of operation is also handled by expansion panels: to select the mode of operation, one must expand the corresponding mode of operation and click the continue button within the expansion panel. Clicking this button will forward the user into the next step.

[18] https://github.com/ReactiveX/rxjs

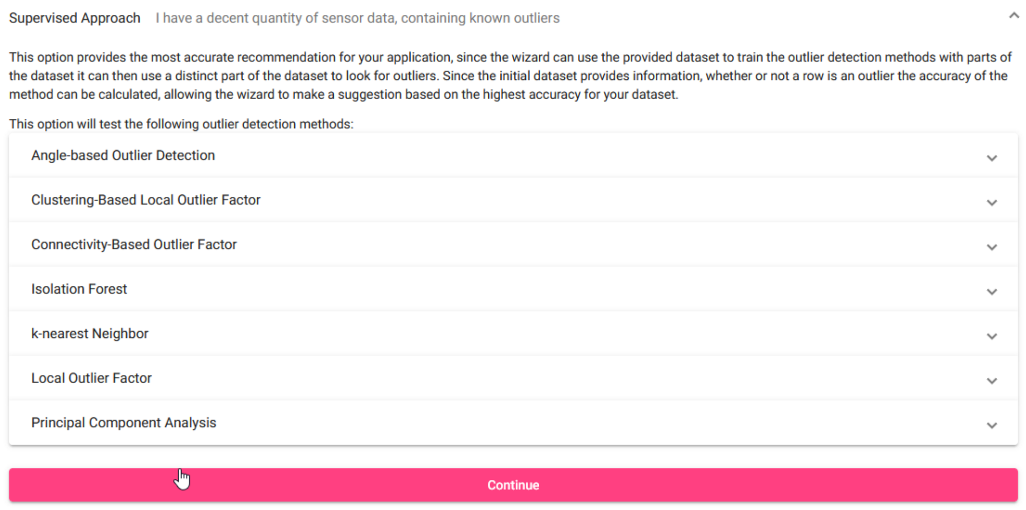


The assisted selection of an outlier detection method requires data.
In the best case, many data sets are available, some of which should be declared as outliers, but a recommendation is also possible if outliers are present.

Please select an option

- Supervised Approach — I have a decent quantity of sensor data, containing known outliers
- Unsupervised Approach — I have a decent quantity of sensor measurements, that contain outliers however the outliers have not been identified as such
- Unsupervised Data Collection Approach — I have no data, but data collection is possible
- None — None of the above applies

(a) The overview, without any expanded option



Supervised Approach — I have a decent quantity of sensor data, containing known outliers

This option provides the most accurate recommendation for your application, since the wizard can use the provided dataset to train the outlier detection methods with parts of the dataset it can then use a distinct part of the dataset to look for outliers. Since the initial dataset provides information, whether or not a row is an outlier the accuracy of the method can be calculated, allowing the wizard to make a suggestion based on the highest accuracy for your dataset.

This option will test the following outlier detection methods:

- Angle-based Outlier Detection
- Clustering-Based Local Outlier Factor
- Connectivity-Based Outlier Factor
- Isolation Forest
- k-nearest Neighbor
- Local Outlier Factor
- Principal Component Analysis

(b) The expanded view for the supervised mode of operation

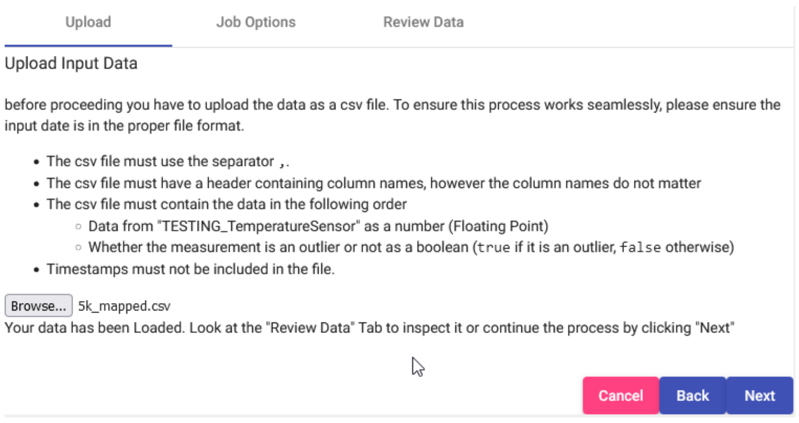
Figure 5.6: The first step of the wizard

5.4.2.2 Step 2: Data Upload and Review

This step combines the Data Upload and Data Review steps initially drawn out in the wireframe diagrams. At first, the user is prompted to upload a csv file, how the file should look, i.e., what columns should be displayed is also shown here, to make sure, one can modify the file accordingly beforehand. It is important to note here that the file will not be transformed by the wizard, since spreadsheet tools like Microsoft Excel[19] or LibreOffice Calc[20] can easily be used to perform these transformations.

File upload, data review and job configuration are part of this step, those subtasks are separated using a tab view. As long as, no file is uploaded, only the data upload tab is active and can be selected. Once the file is uploaded, it is checked, whether a file has the expected number of dimensions and the other tabs get enabled. If the expected number of dimensions mismatches the number of dimensions in the selected file, a warning is shown, informing the user about the potential problem. Since, this limitation does not affect the analysis, it only becomes a problem when deploying the model.

[19] https://www.microsoft.com/en-us/microsoft-365/excel
[20] https://www.libreoffice.org



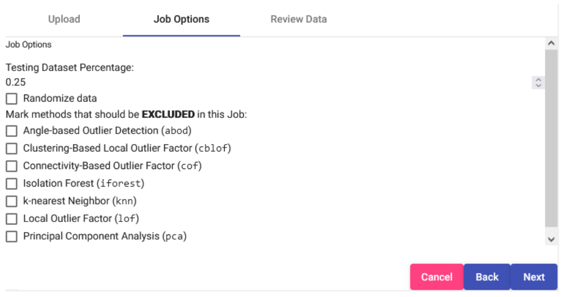
Upload Input Data

before proceeding you have to upload the data as a csv file. To ensure this process works seamlessly, please ensure the input date is in the proper file format.

- The csv file must use the separator `,`.
- The csv file must have a header containing column names, however the column names do not matter
- The csv file must contain the data in the following order
  - Data from "TESTING_TemperatureSensor" as a number (Floating Point)
  - Whether the measurement is an outlier or not as a boolean (`true` if it is an outlier, `false` otherwise)
- Timestamps must not be included in the file.

Browse... 5k_mapped.csv
Your data has been Loaded. Look at the "Review Data" Tab to inspect it or continue the process by clicking "Next"

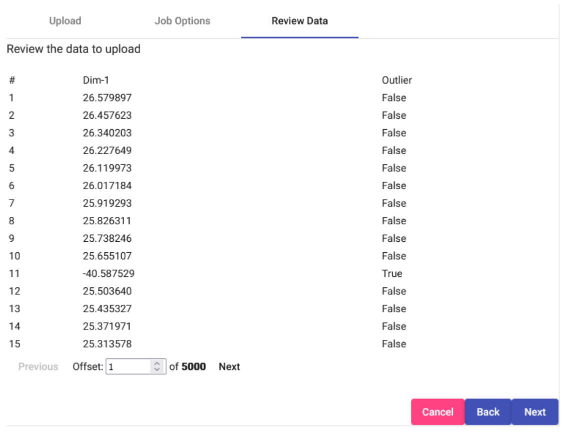
(a) Selection of the input file



Job Options

Testing Dataset Percentage:
0.25
☐ Randomize data
Mark methods that should be **EXCLUDED** in this Job:
☐ Angle-based Outlier Detection (`abod`)
☐ Clustering-Based Local Outlier Factor (`cblof`)
☐ Connectivity-Based Outlier Factor (`cof`)
☐ Isolation Forest (`iforest`)
☐ k-nearest Neighbor (`knn`)
☐ Local Outlier Factor (`lof`)
☐ Principal Component Analysis (`pca`)

(b) Modification of job parameters



Review the data to upload

#	Dim-1	Outlier
1	26.579897	False
2	26.457623	False
3	26.340203	False
4	26.227649	False
5	26.119973	False
6	26.017184	False
7	25.919293	False
8	25.826311	False
9	25.738246	False
10	25.655107	False
11	-40.587529	True
12	25.503640	False
13	25.435327	False
14	25.371971	False
15	25.313578	False

Previous Offset: 1 of **5000** Next

(c) Viewing the contents of the selected file

Figure 5.7: The data upload and review step of the wizard (step 2)

After a file is selected, it is analysed by the frontend application and parsed following the same rules that are used by the Python scripts, performing the analysis, making sure the file can be parsed accordingly, If the file does not exceed a size threshold, it can be previewed in the preview tab, showing 15 rows at once, the modeller can move around in the file using next and back buttons. As well as, the option of specifying an offset from which the rows should be shown. Of course, mechanisms that prevent the setting of invalid offset values, that would either result in viewing data of rows that don not exist, because, the starting index is smaller than the wanted one or ones that would result in a negative starting index, have been implemented. The threshold to view the data in the browser has been set so one megabyte. This value was chosen, since most devices can process this file size in a reasonable time frame.

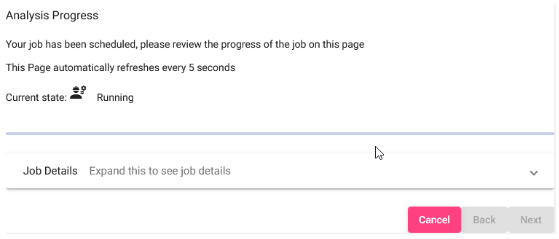
The third tab, is also enabled after the file is selected. It allows the user to explicitly disable several outlier-detection methods, to enable randomization and to set a seed that should be used in order to ensure the dataset is always randomized equally. The option to set a seed only becomes visible, if the data should be shuffled, since this option does not change anything if the data is not shuffled. The percentage of rows in the dataset that should be used for testing can also be changed on this page.

All three parts of this step, are illustrated in Figure 5.7, where a job for a one dimensional model, in the supervised mode of operation, with 5000 rows of randomized sine wave data is about to be started.

Once the user has selected a file, reviewed the data and modified the parameters, if mandatory. The job can be started by clicking the Next button. In the background the UI will take the selected file, upload it to the backend, which returns the file identifier if the upload was successful. This file identifier is then used by the user interface, to create the job according to the predefined parameters. The job is put into the processing queue after creation and the progress of it is shown in the next step, where the user is routed to after the job creation was successful.

5.4.2.3 Step 4: Processing

While the job is pending or in progress, the processing step is shown. It illustrates the current progress using a progress bar. The processing step also has an expandable panel, that contains details about the job, for example, which tasks are already done or if one of them has failed, as a failed task does not automatically result in the termination of the whole job. The detail view also includes an option to terminate the currently running task. Since the page is usually refreshed every five seconds there usually is not need to manually perform a refresh, however an option to do this has been added for the cases, where this may be mandatory. For example if the refresh interval is increased to 30 seconds. The refreshing based on the interval is implemented using a `timer` observable from the RxJS library, that is a core part of the Angular Framework [Dev23a].

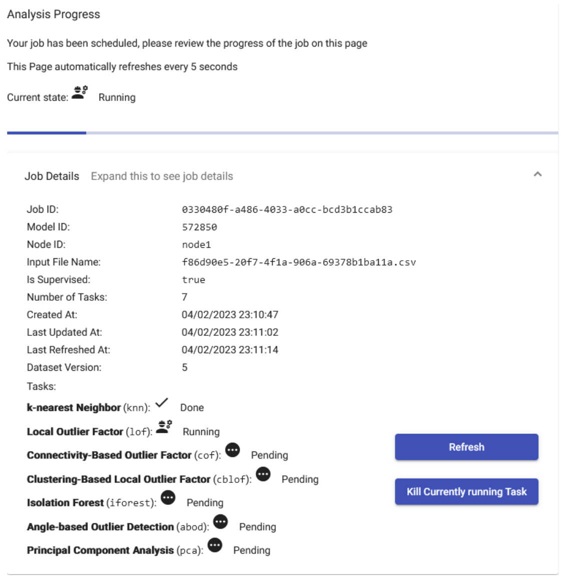

Analysis Progress

Your job has been scheduled, please review the progress of the job on this page

This Page automatically refreshes every 5 seconds

Current state: Running

(a) The overview, without any expanded option


Analysis Progress

Your job has been scheduled, please review the progress of the job on this page

This Page automatically refreshes every 5 seconds

Current state: Running

Job Details Expand this to see job details

Job ID:	`0330480f-a486-4033-a0cc-bcd3b1ccab83`
Model ID:	`572850`
Node ID:	`node1`
Input File Name:	`f86d90e5-20f7-4f1a-906a-69378b1ba11a.csv`
Is Supervised:	`true`
Number of Tasks:	7
Created At:	04/02/2023 23:10:47
Last Updated At:	04/02/2023 23:11:02
Last Refreshed At:	04/02/2023 23:11:14
Dataset Version:	5

Tasks:

k-nearest Neighbor (`knn`): ✓ Done

Local Outlier Factor (`lof`): Running

Connectivity-Based Outlier Factor (`cof`): ● Pending

Clustering-Based Local Outlier Factor (`cblof`): ● Pending

Isolation Forest (`iforest`): ● Pending

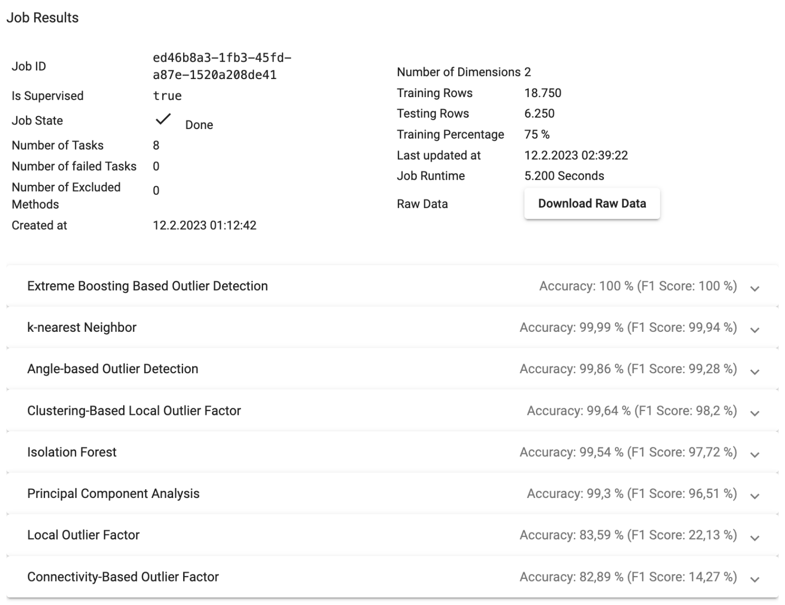
Angle-based Outlier Detection (`abod`): ● Pending

Principal Component Analysis (`pca`): ● Pending

(b) The expanded view for the supervised mode of operation

Figure 5.8: The window shown while a job is being processed (step 3)

The current job can also be cancelled using the common cancel button. Clicking it will tell the backend to cancel the job and to close the wizard without instantiating the validator.



Job Results

Job ID	`ed46b8a3-1fb3-45fd-a87e-1520a208de41`
Is Supervised	`true`
Job State	✓ Done
Number of Tasks	8
Number of failed Tasks	0
Number of Excluded Methods	0
Created at	12.2.2023 01:12:42

Number of Dimensions	2
Training Rows	18.750
Testing Rows	6.250
Training Percentage	75 %
Last updated at	12.2.2023 02:39:22
Job Runtime	5.200 Seconds
Raw Data	Download Raw Data

Extreme Boosting Based Outlier Detection	Accuracy: 100 % (F1 Score: 100 %)
k-nearest Neighbor	Accuracy: 99,99 % (F1 Score: 99,94 %)
Angle-based Outlier Detection	Accuracy: 99,86 % (F1 Score: 99,28 %)
Clustering-Based Local Outlier Factor	Accuracy: 99,64 % (F1 Score: 98,2 %)
Isolation Forest	Accuracy: 99,54 % (F1 Score: 97,72 %)
Principal Component Analysis	Accuracy: 99,3 % (F1 Score: 96,51 %)
Local Outlier Factor	Accuracy: 83,59 % (F1 Score: 22,13 %)
Connectivity-Based Outlier Factor	Accuracy: 82,89 % (F1 Score: 14,27 %)

Figure 5.9: The result view for a job in supervised mode (step 4)

Once the job is finished, the next button becomes enabled and the user can continue into the next task. In case the job is long running, for example when working with a large dataset it is also possible to close the wizard by clicking outside of its modal window, this will not instantiate the validator but will also not cancel the job. If the Wizard it opened again, one can navigate back to the in progress step or the results view step, depending on the fact that job is done at the point of loading.

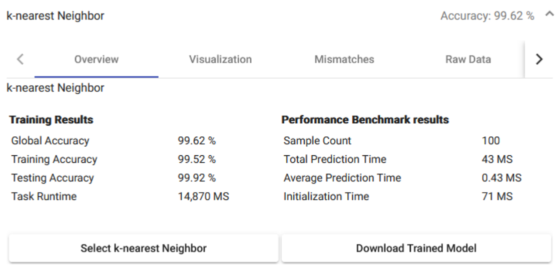
Both the simplified and the detailed progress view are depicted in Figure 5.8. Those figures show a job that is currently running and the frontend waits for the completion of the job.

5.4.2.4 Step 5: Result Presentation

The core of the wizard is the result presentation step that displays the result of the analysis. The view can be split into two sections: (1) the job details and (2) the task details.

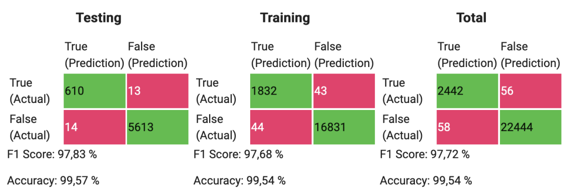
The job details shows data that is either identical in every task, like the number of total rows, the number of rows used for training and testing, the creation timestamp of the job, the mode of operation and the excluded outlier detection methods. There is also an option to download the input file. This job detail section is on the top of the results view, above the tasks details. As illustrated in Figure 5.9.

Figure 5.9 does not include the convolution matrices and the binary classification, however the binary classification is also available for every supervised job in a separate tab, here the number of true and false inliers or outliers are illustrated in a two by two table. By hovering over the cells further values can be displayed. This table is shown three times for the testing and training portion of the dataset as well as one for the whole dataset. This is illustrated in Figure 5.10.



Training Results		Performance Benchmark results	
Global Accuracy	99.62 %	Sample Count	100
Training Accuracy	99.52 %	Total Prediction Time	43 MS
Testing Accuracy	99.92 %	Average Prediction Time	0.43 MS
Task Runtime	14,870 MS	Initialization Time	71 MS

(a) Overview, showing the most important metrics



Testing

	True (Prediction)	False (Prediction)
True (Actual)	🟩 610	🟥 13
False (Actual)	🟥 14	🟩 5613

F1 Score: 97,83 %

Accuracy: 99,57 %

Training

	True (Prediction)	False (Prediction)
True (Actual)	🟩 1832	🟥 43
False (Actual)	🟥 44	🟩 16831

F1 Score: 97,68 %

Accuracy: 99,54 %

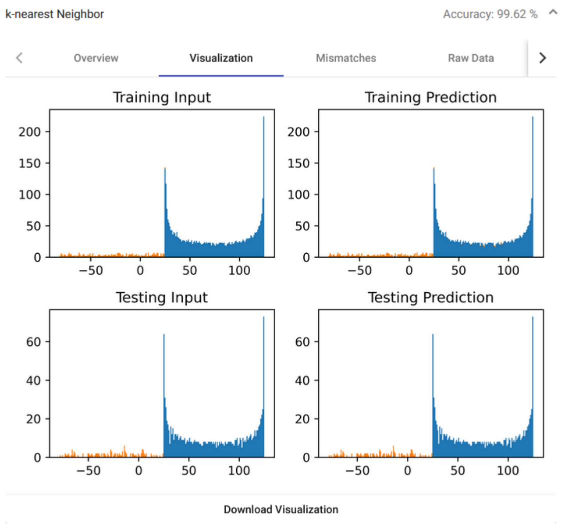
Total

	True (Prediction)	False (Prediction)
True (Actual)	🟩 2442	🟥 56
False (Actual)	🟥 58	🟩 22444

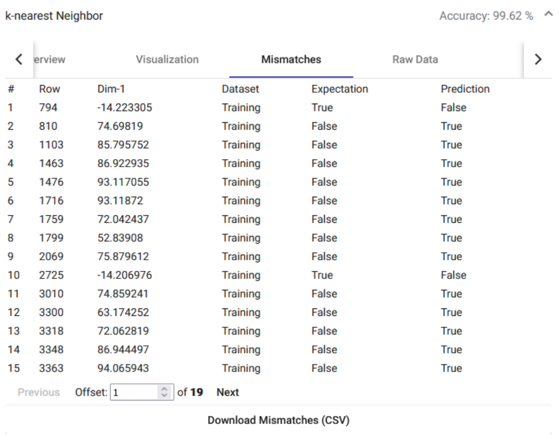
F1 Score: 97,72 %

Accuracy: 99,54 %

(b) The Binary classification view



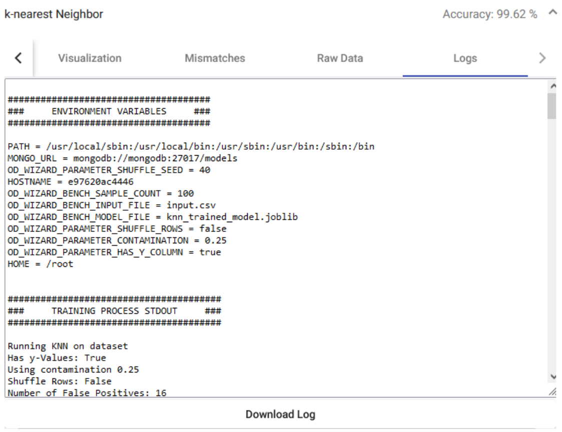
(c) The Visualization



#	Row	Dim-1	Dataset	Expectation	Prediction
1	794	-14.223305	Training	True	False
2	810	74.69819	Training	False	True
3	1103	85.795752	Training	False	True
4	1463	86.922935	Training	False	True
5	1476	93.117055	Training	False	True
6	1716	93.11872	Training	False	True
7	1759	72.042437	Training	False	True
8	1799	52.83908	Training	False	True
9	2069	75.879612	Training	False	True
10	2725	-14.206976	Training	True	False
11	3010	74.859241	Training	False	True
12	3300	63.174252	Training	False	True
13	3318	72.062819	Training	False	True
14	3348	86.944497	Training	False	True
15	3363	94.065943	Training	False	True

Previous Offset: 1 of **19** Next

(d) The list of mismatches



```
#####################################
###     ENVIRONMENT VARIABLES     ###
#####################################

PATH = /usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
MONGO_URL = mongodb://mongodb:27017/models
OD_WIZARD_PARAMETER_SHUFFLE_SEED = 40
HOSTNAME = e97620ac4446
OD_WIZARD_BENCH_SAMPLE_COUNT = 100
OD_WIZARD_BENCH_INPUT_FILE = input.csv
OD_WIZARD_BENCH_MODEL_FILE = knn_trained_model.joblib
OD_WIZARD_PARAMETER_SHUFFLE_ROWS = false
OD_WIZARD_PARAMETER_CONTAMINATION = 0.25
OD_WIZARD_PARAMETER_HAS_Y_COLUMN = true
HOME = /root


#######################################
###     TRAINING PROCESS STDOUT     ###
#######################################

Running KNN on dataset
Has y-Values: True
Using contamination 0.25
Shuffle Rows: False
Number of False Positives: 16
```

(e) The log file

Figure 5.10: The detailed results for one task, when running in supervised mode (step 4)

The lower section contains an expandable panel for every task, i.e., every outlier detection method that was executed. If the job is a supervised job, the expandable panels are sorted based upon the achieved accuracy for every method, assuming accuracy is the only metric that matters, the best suited outlier detection method is shown on first and the one with the worst accuracy is at the bottom of the list, before the failed tasks, which are always at the very bottom of the list. The back end does not create a task for excluded method and therefore they are not shown in the list of tasks.

The expanded task panel contains a tab panel illustrated in Figure 5.10, of which the initial tab shows an overview, presenting the metrics for the task, like accuracy and the prediction times. On this view one is also able to select the expanded method to instantiate the validator, if the model produced by `joblib` does not exceed the size of approximately 16 megabytes one can also download this file, to use this model for further processing or analysis, without having to retrain the model.

The other tabs can be used to view the visualization, the list of the mismatches the list of all predictions of the input file and the log file produced by the backend, useful to potentially identify any reasons why a task may fail. The visualization and the mismatches tabs are optional tabs, since they are only available if certain preconditions have been met. In case of the visualization the condition is bound to the number of input dimensions, since a visualization is only produced for one and two dimensions, if one for example performs an analysis on a three dimensional model the visualization is not shown. The mismatches list is only shown if the analysis was able to compute the mismatches, i.e., when the job was running in the supervised mode of operation.

If an analysis of an unsupervised job is shown the tasks are not sorted, apart from placing failed tasks at the very bottom, since there is no metric to compare them against. Of course the accuracy is also not shown in that case.

5.4.2.5 Job List

The job list step is a step that is only displayed if there is at least one existing job for a node, it represents a paginated list that allows one look at existing jobs or to create a new one. If a new one is created the user gets redirected to the first step and if an existing job is opened the user is either redirected to the results view or the progress view, depending on the current state of the job. One can view failed, pending/running and done jobs. Cancelled jobs cannot be opened, however they are still included in the list.

5.5 Generating test data

In order to test the implementation and also to perform an evaluation on the usefulness of the wizard a method to get data that fulfils the requirements needed for our analysis had to be found. Since there was almost no real world data available the easiest solution was the implementation of utilities to generate random data, that contains a predefined percentage of outliers.

To achieve this several types of random data generators have been implemented in Go(lang)[21], the choice of programming language is not based upon any objective criteria, instead the decision to implement these generators in Go was only based upon personal preference and past experiences with the language.

All of the generators produce labelled data, however the labels can be easily stripped later using a simple find and replace procedure, for example by using the Stream Editor (`sed`)[22]. The generators always have at least two ranges,the inlier range ($R_{Inliers} \in \mathbb{R}$) and the outlier range ($R_{Outliers} \in \mathbb{R}$), in some cases multiple subranges for either outliers or inliers are used, for example the inliers may be generated in the value range between 0 and 50 as well as 100 to 120, for example. For simplicity this is still represented as one set. All generators always make sure that an outlier is within the outlier range but not within the inlier range, even if the inlier range is a subset of the outlier range. Mathematically speaking the generator makes sure the following condition is fulfilled for any outlier ($v_{Outlier}$): $v_{Outlier} \in R_{Outliers} \wedge v_{Outlier} \notin R_{Inliers}$. Of course it is also ensured that the opposing condition $v_{Inlier} \in R_{Inliers} \wedge v_{Inlier} \notin R_{Outliers}$ is always fulfilled as well.

When working with multiple dimensions, the generated values are based upon the same constraints for every dimension, however it is made sure that all dimensions get different values. In the multidimensional mode the tools also generate multiple kinds of outliers, outliers that only affect one dimension and outliers that affect multiple dimensions. How many dimensions are affected by the outlier is chosen at random, but can be configured using command line flags, just like other input parameters, such as the number of rows and dimensions. Generator specific parameters are also set this way. Every generator has a built-in help command that lists all possible options. If no parameters are provided a two dimensional file with 10,000 rows is generated. The generated files are always written to the command line, to either allow further processing, for example to remove the labels or to store them using stream redirection, assuming a UNIX-like/based operating system is used to build the dataset.

The following data generators have been implemented:

- The simplest generator is the random generator it generates completely random data, based upon two predefined ranges: the inlier range and the outlier range. Apart from fulfilling the aforementioned conditions all values are generated completely at random.
- The sine wave data generator produces inliers using a sine function, of which the amplitude and the offset is predefined to determine the minimum and maximum value of the function. To ensure an offset between all dimensions a random offset for the index is chosen for every dimension, before generating the data. An option to add some distortion, by adding a small random value to the sine output has also been added. With this generator outliers are also just generated at random, identical to the random generator.
- The random cluster generates a random amount of value clusters with a random value as a starting point for every dimension, when a new random inlier is generated first the cluster for which the data should be generated is chosen then the spread from the base offset is randomly chosen, resulting in multiple hotspots of data. Outliers are generated at random, identical to the other options.

[21] https://go.dev/

[22] https://pubs.opengroup.org/onlinepubs/009695399/utilities/sed.html

6 Evaluation of the wizard

To investigate the use of the wizard developed for suggesting outlier detection methods, an evaluation has been designed and performed, of which the design will be presented and the results get discussed in this chapter, by first drafting, what the goals of the evaluation are, how it should be conducted. Afterwards, the final design of the evaluation will be discussed, before investigating the results of the evaluation.

6.1 Drafting the evaluation

The main goal of the evaluation is to find out how useful the implemented wizard is to people that represent the intended users of the wizard.

Usefulness in itself is very general, and just considering the question how useful the tool is, is not specific enough, since we want to find out, which features are useful and what may be improved in a revised version of the wizard.

The main focus of the evaluation is on the wizard itself, as well as the metrics and visualizations it uses in order to help the user to make the decision, which outlier detection method is best suited for a dataset.

With these base goals defined, we identified the the following questions that should be answered by the subjects of this research:

- How well is the wizard integrated into the IoT Application Modelling Tool?
- Does the tool provide value, when assisting the user on the topic of choosing an outlier detection method?
- Do the provided visualizations for both one and two dimensional inputs provide a value?
- Are the metrics provided sufficient for making a good decision?

In order to answer these questions, it is mandatory that the subjects are getting familiarized with the tool. For this, the tool must either be shown to the subject by the use of a presentation or by actually using it. In order to make sure the subjects get a hands on experience when working with the tool, it makes more sense that the subjects try the tool out by themselves, in order to make the experience more hands-on instead of making an assessment by watching a demo, especially with regard to questions of intuitiveness.

The study conducted will be based upon the procedure described in the following:

1. First, the participant is shown a short instruction on how to get to the assistant and how to initiate a process. The main intention is to give the candidate a rough overview of the process. This is done to ensure that the candidate can get to the parts of the wizard that needed to be looked at in order to assess the above points without further guidance. Keeping this to a minimum is also an important consideration, to make sure the candidate can rate the intuitiveness of the wizard.
2. After the brief introduction, the candidate is intended to explore the tool on his own, to take a look at the capabilities of the wizard. As the wizard requires datasets to be used, a number of datasets have been provided for one, two and three dimensional setups.
3. The user will be asked to rate his experience in the aforementioned fields using a grading system.
4. After the rating, the participant is asked to name aspects that he or she thinks should be improved in regards to the visualization, the metrics and the user experience.

This evaluation design is both suitable for a survey and an interview based evaluation. This is intentional, since the final decision still had to be made in this early drafting stage of the evaluation. The decision was later made to conduct the evaluation as a survey.

6.1.1 How should the evaluation be done? Using a survey or interviews?

An important consideration in designing this evaluation is the way it is meant to be performed: Either as a survey or by performing interviews and either face-to-face or virtually. The decision was made to conduct the evaluation as a survey, even though an interview-based process would probably provide better value when investigating the usability and intuitive of the application. Tools like screen recordings cloud have also been used, in order to measure the time needed to perform certain tasks like opening the wizard and starting an evaluation.

The main reasons why the decision was mad eagainast an interview-based evaluation are mostly of organizational kind, the main benefit of a survey is the fact that it can take place in an asynchronous matter, meaning no one has to perform or supervise the interview, since it is done in a self-service fashion. A survey has the potential to get more candidates, since a survey, even with the exploration part, usually does not take as long as an interview. In terms of analysis, a survey is also much more efficient, as the results can be analysed directly using tools such as Google Forms[1] or LimeSurvey[2]. It is not necessary to transcribe the interviews in order to analyse them.

6.1.2 Providing data for the evaluation

As discussed in Step 2 above, datasets are provided to ensure the evaluation focusses on the primary objective of the wizard: assisting the modeller in the selection of an outlier detection method. Most of the provided datasets are randomly generated using the data generators discussed in Section 5.5. However, it was also possible to provide a real-world dataset, derived from indoor temperature

[1] `https://www.google.com/forms/about/`
[2] `https://www.limesurvey.org/de`

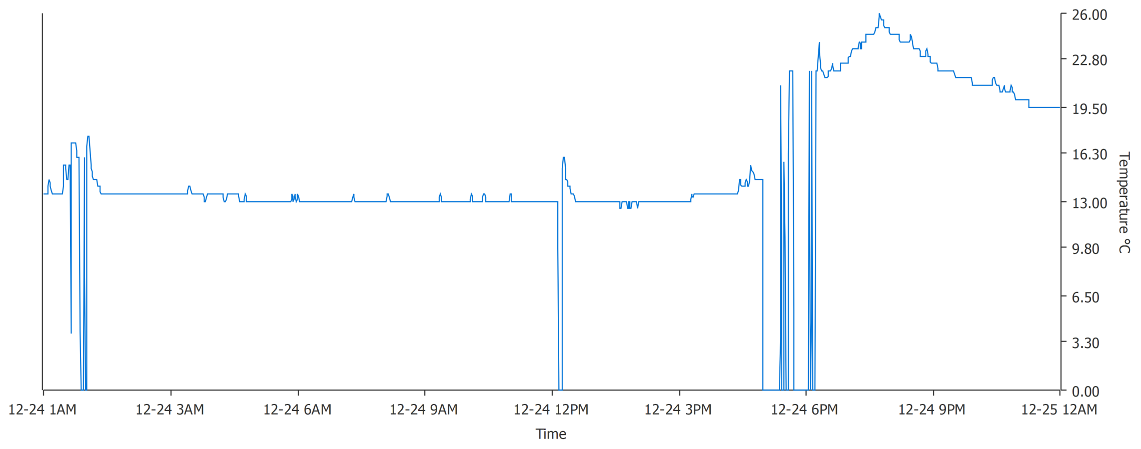


Figure 6.1: Sample of the real world dataset provided in the evaluation

measurements collected by a wireless smart home power plug (AVM Fritz!DECT 200[3]) used to monitor the activity of an electronically controlled pellet stove. Due to external factors, like connection losses to the base station due to the relatively large distance between them, empty readings were read by the collection program[4], resulting in a value of zero being collected. These zero readings are considered outliers in this scenario. A data sample illustrating a comparably large number of outlier measurements is shown in Figure 6.1.

6.1.3 Providing an instance of the wizard

Getting a hands-on experience with a rather complex deployment process to get the application up and running is a problem. The best solution for this, is the usage of a pre-deployed instance. To achieve this, an instance of the IoT Application Modelling Tool was deployed on a virtual machine provided by the Universtty of Stuttgart. However, this Virtual Machine is only accessible from within the network of the university. To also allow persons from outside the university to participate, an option to make the deployment as easy as possible had to be found. As previously discussed, Docker Compose is intended to be used to make the deployment of tea application as simple and standardized as possible, making it perfect for this purpose. With Docker Compose the application can be deployed locally by running one command, assuming the prerequisites, i.e., installing a Linux capable Docker runtime, are fulfilled. To make this process as simple as possible a quick and simple set up guide has been written, that is also intended for use if there are problems with the hosted instance.

Some of the outlier detection methods implemented are quite time consuming. To make sure a job does not take 15 minutes or more to complete, some time consuming outlier detection methods have been disabled on the hosted instance.

[3]https://avm.de/produkte/fritzdect/fritzdect-200/

[4]How the data for this real world data was collected in detail can be seen in Appendix A.1.

For the evaluation, a snapshot of the implementation was taken, on which the evaluation was performed upon. As a result, some improvements suggested by participants have been implemented while the survey was in progress. This was done to ensure the results of the evaluation are based upon a common basis, making the results of the evaluation comparable, even though some things that may have influenced the ratings by participants have already been implemented, potentially increasing the ratings.

6.2 Designing the survey

As explained above, surveys can be carried out using an online tool, which also makes the analysis very fast and effective. The tool used to conduct the survey is Google Forms. Here, the questionnaire for the survey can be designed using a WYSIWYG Editor (What you see is what you get).

Now that the evaluation has been outlined, a questionnaire can be developed to serve as the basis for the survey. To make the survey more understandable, some additions to the previously outlined concept had to be made. The initial tutorial to teach the basic functions of the assistant is displayed using a text tutorial, after an instruction text, that describes the goal of the survey, who is responsible for the survey, where the provided instance can be accessed and where the participant can find the sample data for performing the tutorial. A video tutorial was also an option. However, the decision was made against it, since most of the steps are illustratable using some pictures. Viewing a video is likely to take more time than just scrolling through the tutorial. To get an impression on how the tutorial looked within the survey, it is depicted in Figure A.8.

After the tutorial, the user should open the application and try the wizard. The participant is intended to rate statements regarding the aspects described above based on six grade scale. The scale goes from --- (Very bad) to +++ (Very good) over -- (Bad), - (Slightly bad), + (Slightly good) and ++ (Good). An option to rate for neutral has been excluded explicitly, since the participant should still give a rating to indicate in which direction the participant thinks the rating should go. Another reason is that a neutral option is often chosen as the dumping option, that gets chosen if someone is insecure about rating the application, as discussed in a blog article by Chyung [Chy19].

The subject is asked to rate the following statements in the given order:

- S_1 The wizard is intuitive to use overall.
- S_2 The wizard in well integrated into the IAMT.
- S_3 The tool does provide value, when assistance for selecting an outlier detection method is needed.
- S_4 The visualization of the data with 1 dimensional inputs is sufficient.
- S_5 The visualization of the data with 2 dimensional inputs is sufficient.
- S_6 The provided metrics to determine what's the best suited method are sufficient.

Since the subject was expected to have worked with the tool, we assume that every aspect can be rated by the candidate. Therefore, none of these options can be skipped and there is also no option to express that the participant is not able to answer the question.

Question	+++	++	+	-	--	---
S_1	0	0	0	2	3	0
S_2	0	0	0	0	1	4
S_3	0	0	0	2	3	0
S_4	0	1	2	1	1	0
S_5	0	0	0	1	4	0
S_6	0	2	1	1	1	0

Table 6.1: Survey results of the overall ratings.

After performing the general ratings, the participant is redirected to a new step where the participant can provide optional comments and improvements regarding the application. The following optional text fields for suggestions are shown to the participant:

- What do you think could be improved in regards to the user experience?
- What do you think could be improved in regards to metrics provided for decision-making?
- What do you think could be improved in regards the visualization of the results?
- Are there any outlier detection methods, that could be added to the tool?
- Do you have any other suggestions?

After these options to provide suggestions the participant is asked to provide some information about himself or herself. The main purpose of these questions is to find out how familiar the participant evaluates him or herself in the topics related to this work.
To achieve that the participant is intended to rate himself on a three level grading scale from *No experience* over *Some experience* to *A lot of experience* in Internet of Things, Outlier detection, and the tool stack on which the wizard is built upon, like the MBP and IAMT.

Lastly, the participant is asked to group his profession, in one of these groups of persons: student, researcher, IT professional, lecturer and an option allowing the user to name his profession if it is not fitting one of these groups. Overall, this question was included to potentially determine how valuable the given answers may be, for example, one could expect that a researcher is probably more familiar with the topics from a scientific perspective, where an IT professional, working in industry may asses the tools value based upon its practical applicability. However, after the survey has been conducted this idea turned out to be hard to asses. For that reason this, question was not providing value to the evaluation overall.

6.3 Investigating the results

The survey was opened for a period of two weeks. Within this period, five participants took part. Two of the participants declared themselves as IT professionals and three of them as researchers. In terms of previous experience, all of the participants have had some experience with IoT and three of the five also stated to have some experience in both the fields of outlier detection and the overall tool stack. However, the three persons stating to have some experience with outlier detection were not identical with the ones stating the same about the tool stack.

The overall ratings for the statements are illustrated in Table 6.1. Here, both positives and negatives of the wizard can already be observed. In regards to the integration of the wizard into the modelling tool and the overall intuitivity of it (S_1 and S_2), the ratings are all positive, with some potential to further improve. On the other hand it is clear that the provided metrics and the one dimensional visualization clearly have to be improved, since most of the ratings in these cases are negative, with the more significant problem being the provided metrics. The two dimensional visualization on the other hand also received a relatively positive rating and is therefore perceived to be better than the one dimensional version.

The most interesting thing, in the evaluation are the suggestions and potential problems, that explicitly get called out by the participants. These will be discussed in the following.

In regards to the user experience, the following aspects were mentioned that could potentially get improved:

- Small improvements in the the already existing implementation of the modelling tool, these improvements were:
  - The Login function that allows the user to attempt to open a model even though he is not logged in any more.
  - The order of the options in dialog to create and open a model.
  - The unlabelled switch to add testing operators.
- The lack of a close button seems unintuitive.
- The back and next buttons should be used, for selecting the mode of operation and the target outlier detection method, instead of using buttons within the *description* of the option to select it.
- Descriptions for outlier detection methods are missing, within the wizard.
- The hyper-parameters for outlier detection methods, like the k parameter of the k-nearest-neighbour algorithm, are not listed.
- The usage of *Is Supervised* within the Description of an outlier detection method is confusing.
- The descriptions of the process itself should include more documentation, for example on how the results can be interpreted.
- The exclusion of outlier detection methods, to only perform the analysis on one method is not intuitive.

As this shows, the UI still needs some minor pollishments, like adding a close button, using the next and previous buttons more often and ensuring the algorithm selection is more intuitive. However, the min problem is the lack of descriptions. In some regards this was an aspect that was clear before the evaluation was conducted, since the intended descriptions for the outlier detection methods have not been added. Apart from this known problem some minor things that should be improved have been discovered through this evaluation.

The suggestions regarding the visualizations mostly focus on the one dimensional visualization. Since this one is, as we saw previously, the more problematic one, to make understandable. However, the majority of the criticism revolves around the implementation itself and not the kind of visualization. The suggestions and problems for the one dimensional visualization include;:

- The lack of a legend to label the data and the lacking labels of the x and y axis.
- Outliers are hard to identify if the range between the minimum and maximum values is very large and many data are piled up in one place.

These problems were mentioned by three of the five participants, which shows why the one-dimensional visualisation received such a poor rating. Furthermore, a visualization that compares all outlier detection methods has been suggested as an improvement as well as a bar chart for the two-dimensional visualization.

The most criticised aspect was the selection of metrics. Here, the main complain was the lack of a convolution matrix and the use of the F_1-score to rate outlier detection methods which was not yet implemented in the version that was used to conduct the evaluation. While the evaluation was conducted, this was changed. How the convolution matrix and all the corresponding metrics have been implemented has already been discussed previously. Because this was a major criticism, in regards to the metrics in the supervised mode of operation, it is likely that the ratings for S_6 may become better overall, if the evaluation would be conducted again, with the newer version. However, participants also perceived the provided metrics in the unsupervised mode of operation as insufficient, which could also be the root cause for the bad ratings in regards to S_6. To improve the metrics in the unsupervised mode of operations, further investigations are mandatory.

Apart from that, only one further suggestion was made: Making the change from an instantiated outlier detection validator node back to an uninstantiated one easier. For example, to select another outlier detection method or to perform a new analysis based on another dataset. Currently, this can only be achieved by opening the edit dialog of the node, the change the type of the node back to the uninstantiated type. This is unintuitive and may, for example, be improved by adding another option for faster modification instead.

In terms of the provided outlier detection methods, no suggestions have been made.

The evaluation has shown, that the tool does provide value, but still needs some fine tuning in regards of its overall usability, the metrics in the unsupervised mode of operation and the one dimensional visualizations in order to further improve the usability in of the wizard.

6.4 Conclusions from the Survey

From this evaluation we can conclude, that the proposed wizard has been well integrated into the IoT Application Modelling Tool, since all participants rated this criteria with at least ++ (Good). In addition, no one has made any additional suggestions or complaints in regards to that. The result for intuitiveness was slightly worse, with most participants voting for ++ (Good), but the rest only for + (Sightly good). Also in this respect, suggestions for improvement were made, such as adding a button to close the wizard or the permanent use of the Next, Back and Cancel buttons, instead of occasionally resorting to other buttons to get to the next step.

However, the evaluation also showed that there are things that need to be improved. In addition to the visualizations, especially the 1D visualization, the metrics that are provided were also criticized. Apart from the metrics, provided in the unsupervised mode of operation, another major point of criticism was the lack of more expressive metrics, in the supervised mode of operation, were requested, potentially having a significant influence on the negative ratings of the statement S_6. To get a real measurement for the newer version, that includes these metrics, the survey would have to be conducted again. In this case, a larger number of participants should be aimed at in order to obtain more opinions and to consult further expert groups that may have a different perspective on the topic.

7 Related Work

Both Al et al. [AMM+21] and Jiang et al. [JHSG+20] review and investigate different methodologies for outlier detection in the context of the Internet of Things, while they do not propose a suggestion mechanism the approaches reviewed in these publications they do investigate other approaches to outlier detection in an IoT environment, building on top of the outlier detection methods used in this work.

The pyOD library proposed in the work by Zhao et al. [Zha23; ZNL19] provides implementations for roughly 40 different outlier detection methods. The methods have been implemented with attention to scalability and robustness, through the generic interface of the methods it is also possible to build abstract systems where the method can vary. Examples of this are the analysis and benchmark steps we implemented. The library is implemented in Python.

Many of the methods implemented in pyOD are evaluated in a work by Han et al. [HHH+22a; HHH+22b] with the main goal to investigate a set of 30 outlier detection methods in regards to their performance[1], based on analysing different kinds of anomalies and when working with different levels of supervision. The analysis done in this benchmark is based on a set of different datasets, containing different kinds of anomalies. Unlike pyOD, the results of this work are not used in this work. However, this paper is certainly interesting, when selecting the set of outlier detection methods. That could be added to the suggestion wizard.

Some outlier detection methods have hyper-parameters that can be adjusted, to potentially improve the performance of an outlier detection method. However, adjusting these parameters manually is not an easy task and requires a certain degree of knowledge in regards to the outlier detection method for which the hyper-parameters shall be modified. To counter this problem Xu et al. [XKC19] propose an approach that automatically configures the hyper-parameters of the Local Outlier Factor outlier detection method based on a heuristic method. This work could contribute to an improvement to the performance of the local outlier factor detection method, within the suggestion algorithm, since hyper-parameters are currently completely omitted, as the wizard uses the default values from pyOD.

The MetaOD approach proposed by Zhao et al. [ZRA20] is an automated approach that suggests an unsupervised outlier detection algorithm, as well as it's meta-parameters, based on a meta-learning approach that trying to make a suggestion based upon past performances of outlier detection methods on other datasets. The goal of the work of Zhao et al. is very similar to the core goal of this work, i.e., finding a well suited outlier detection method for a given dataset. However, unlike the approach that is proposed in this work, it can be fully automated, with the drawback that it does not provide

[1]Unlike in the rest of this work performance in this case does not mean the time needed perform a prediction, instead it should be considered a much more broad term here.

a lot of transparency on how the decision has been made. Integrating MetaOD into the wizard proposed in this work, could still represent a large improvement in usefulness of the wizard. Due to that, the integration of MetaOD into the wizard is discussed in Section 8.1.

Another interesting approach to perform outlier detection in a time-series based environment, proposed by Lai et al. [LZW+21] called Time-series Outlier Detection System (TODS). TODS is a platform that handles end-to-end processing of time series datasets in order to detect outliers, to achieve this the flow of data can be modelled using a directed graph that represents the flow of data. Similar to the previously mentioned MetaOD it also provides a mechanism to automatically determine the best suited detection flow for a given dataset.

8 Conclusion and Outlook

The main goal of this work was the investigation of automated validation in a model-driven IoT environment, i.e., filtering out measurements or data, considered abnormal, at a point where the modeller thinks this is mandatory. In order to achieve this, without having to manually model the validation, for example by defining rules, which can be used to determine if a value is normal or abnormal. This work identified, that machine-learning based outlier detection methods are a great fit for this problem, since they can be trained. In order, to become suited for any environment, reducing the coupling between the model and the environment, in turn potentially making the model more reusable. The only drawback, is the requirement of existing sensor data, either labelled, i.e., declaring if a value is an outlier (an abnormal value) or unlabelled, where this information is missing. However, it was also shown, that the selection of a suited machine-learning based outlier detection method is not an easy process, as there are many different approaches attempting to solve this problem, some better suited for a given scenario than other ones. To find the best suited outlier detection method, metrics and potential approaches to visualize the predictions of an outlier detection method have been identified. With the main goal of making the outlier detection methods comparable, within the context of one model, assisting the modeller in the decision of selecting the best suited outlier detection method.

From there a wizard with the goal integrating the ability to run and compare outlier detection methods, in the already existing IoT Application Modelling Tool has been designed. Here we saw that, a new kind of node had to be introduced to the model: the validator node, that acts as a filter in the pipes and filters model. We also saw that this new kind of node introduces many new problems, in regards to how the validation should be executed during runtime, here three different options, each with a its pros and cons. The methods varied from a completely centralized approach in which every validation nodes tasks would be executed on just one device to a fully decentralized option where the tool stack tries to run the validation on existing nodes at all cost, i.e., the validation is either performed on the source or target device, depending on the model, the introduction of a new node, that may have to be deployed on another device should be avoided at all cost. Another approach, that is most likely, the best suited one is a middle-ground between the two, where the validation can be deployed on any available device, but a suggestion is made, that aims to keep the number of newly introduced devices as low as possible.

After conceptually evaluating the whole flow, from analysis to aspects of the execution, a portion of the previously proposed concept has been implemented: the suggestion wizard, which was implemented as an extension of the IoT Application Modelling Tool, with this extension, the modeller can now select a outlier detection node, model the data-flow tan and from it. To initiate the wizard, which will take a given dataset, containing measurements for every inbound edge. This data is then used to train the outlier detection methods, attempting to make them comparable, so that the modeller can select the one that seems to be best suited for the model, based on either the metrics and the visualizations. Depending on the mode of operation, the selection of an outlier detection method could be done completely automatic, in case the data is labelled and assisted in case of unlabelled data. The reason for this is the lack of measurable objective metrics on the performance of an outlier detection method, in case the data is unlabelled, apart from the prediction time, which cannot be used to determine, whether or not, the predictions made by the outlier detection method are likely to be correct or not. In case of labelled data, metrics, like the accuracy or the F_1-score can be used to objectively identify the outlier detection method, that performs best for the given data, making an automated selection possible. However, the current implementation of the wizard still requires the modeller to select the outlier detection method he wants to use in both modes of operation.

The implementation of this wizard has been evaluated, during a survey targeting persons, that are familiar with the topics, this work is involved in, has shown that the wizard does provide value, is intuitive to use and has been integrated well into the IoT Application Modelling Tool. However, it has also shown, that the wizard still has room for improvement, especially in regards to the metrics of the unsupervised mode of operation. The evaluation has also shown, that the visualizations provided can also be improved, to further improve the experience.

Overall, this work provides a basis for further improvements, such as the support for more outlier detection methods, or mechanisms to get better metrics in case the data provided is unlabelled.

8.1 Outlook

While this work presents an approach to choose a well suited outlier detection method, it does not provide an implementation of the whole flow, to make the instantiation of the selected outlier detection method possible. This section will therefore investigate further mandatory steps, to allow this. It will also investigate, how the proposed approach could potentially be further improved.

8.1.1 Improve the wizard based on the suggestions from the evaluation

The evaluation has shown that our wizard works and also has the capability to provide value. However, it has also shown that there are many things that should be improved upon, such as the overall documentation during the process and the documentation of the outlier detection methods themselves.

Furthermore, the visualizations can be improved, by adding labels and a legend to show what colour represents which portion on the dataset to the one-dimensional visualization. The alternative approach to visualize the data, based on a time-series diagram could be implemented to also see outliers in that manner, this would also enable a type of visualization for a arbitrary amount of dimensions, since the dimensions are split in a time-series diagram. Another option could be a view that combines all visualizations of all tasks at once, in order to directly compare the produced results side-by-side.

8.1.2 Improving the suggestion mechanism for unlabelled data

One main suggestion in the evaluation was the lack of metrics in the unsupervised mode of operation. Here one could consider a feature that allows the comparison of the outlier detection methods by setting one of them as the truth. This would allow the comparison between outlier detection algorithms.

Another promising idea is the MetaOD approach proposed by Zhao et al [ZRA20], which has been introduced briefly in Chapter 7. It could be used to get a potentially well suited outlier detection method for a dataset with unlabelled data. In which the wizard currently only supports the selection based upon human decision. Furthermore one could investigate the usability of Time-series Outlier Detection System (TODS) proposed by Lai et al. [LZW+21] for this use-case.

8.1.3 Collecting data for making a suggestion

One of the main goals of this work was the implementation of the wizard based upon existing data. However, in the concept, an approach has been proposed that would allow the collection of data by deploying the portion of the model relevant for training the outlier detection model. To achieve this a more technical concept must be created and implemented, especially investigating how status information will be transferred from the collector node to the modelling tools backend. To see progress information and to enable the download of the collected data to perform the analysis.

8.1.4 Implementing the instantiation and runtime of models with validators

This work also investigated how validation nodes can be distributed within the topology of the model. Three different approaches were proposed: the fully centralized validation, the fully decentralized validation or a combination of both. It requires the user to determine where the validation should happen, while giving a suggestion intended to keep the number of newly introduced nodes as low as possible.

Further work must determine the validation approach and the implementation of the validation component as well as the adjustment of the ME-Configurator and the Messaging Engine. The modelling tool will also require further modification, as the tool itself does not support the instantiation of validators in its current state.

8.1.5 Investigating a semi-supervised approach

In Section 2.2.1, three different machine-learning based outlier detection methods, have been identified: Supervised, semi-supervised and unsupervised methods. While most of the outlier detection methods used by the wizard in this work are based on unsupervised learning, the use of labelled data is still a better option, since binary-classification-based metrics can be derived from it using this approach. With the idea of a semi-supervised algorithm in mind, meaning that the input data is only partially labelled by either identifying inliers or outliers, it may still be possible to find metrics that allow the comparison of outlier detection methods. For example, the accuracy for the subset could still be computed, since the number of correct and incorrect predictions can be counted. By implementing this, a third mode of operation that is based on existing data could be included.

8.1.6 Adding more outlier detection methods

While the set of currently implemented outlier detection methods covers a broad range of different approaches for machine-learning based outlier detection, there are many more outlier detection methods that could be implemented. On the one hand, there are outlier detection methods that can be considered improvements or slight modifications of the already implemented outlier detection methods such as the median or average based k-nearest Neighbors algorithm that slightly modify the original k-nearest neighbour algorithm [AP02; RRS00]. On the other hand, further outlier detection methods, that may be considered interesting for the evaluation using the wizard, could be implemented.

8.1.7 Add an option to choose meta parameters

Some outlier detection methods implemented have meta parameters, like the k in the k-nearest neighbour outlier detection method. To further improve the suitability of outlier detection methods, adding an option to either modify this value or to introduce multiple tasks with the same method but different parametrises could further increase the use of the tool. With this, one should however still keep it as simple as possible, in case the modeller either does not know what these meta parameters mean.

8.1.8 Supporting larger files

The current implementation of the outlier detection wizard has one huge limitation: the application currently cannot store files that exceed the maximum document size of MongoDB at roughly 16 MB. While this is not a significant problem when storing the visualizations and raw data files, it does become a problem if the `joblib` Model files should also be stored in the database, for later use, e.g., the model instantiation in the Validation Engine [Mon23]. Some options to solve this issue could be the use of the file system for large files or the use of an object storage, like MinIO[1] or OpenStack Swift[2], for files that exceed a certain threshold. Instead of storing it directly in MongoDB as part of the document, a pointer is stored that can then be used to retrieve the data.

8.1.9 Evaluating the usability and usefulness on a larger scale

The number of things that could be improved can be considered almost endless, if the considerations are taken into account wide enough. However, apart from the, technical improvements the evaluation could be performed again on a larger scale. In order to further evaluate the usability of the tool an experiment could get conducted, where the user has to perform certain tasks, without getting a guide on how the wizard actually works.

[1] https://min.io/
[2] https://wiki.openstack.org/wiki/Swift

Bibliography

[AMM+21] R. Al-amri, R. K. Murugesan, M. Man, A. F. Abdulateef, M. A. Al-Sharafi, A. A. Alkahtani. “A review of machine learning and deep learning techniques for anomaly detection in IoT data”. In: *Applied Sciences* 11.12 (2021), p. 5320 (cit. on pp. 31, 83).

[AP02] F. Angiulli, C. Pizzuti. “Fast outlier detection in high dimensional spaces”. In: *European conference on principles of data mining and knowledge discovery*. Springer. 2002, pp. 15–27 (cit. on pp. 31, 88).

[Bae22] Baeldung. *F-1 Score for Multi-Class Classification*. https://www.baeldung.com/cs/multi-class-f1-score. 2022 (cit. on p. 20).

[Bib22] Bibin Wilson and Shishir Khandelwal. *How to Reduce Docker Image Size: 6 Optimization Methods*. https://devopscube.com/reduce-docker-image-size/. Jan. 2022 (cit. on p. 63).

[BKNS00] M. M. Breunig, H.-P. Kriegel, R. T. Ng, J. Sander. “LOF: identifying density-based local outliers”. In: *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*. 2000, pp. 93–104 (cit. on p. 31).

[Bri23] Brian Troncone. *Learn RxJS*. https://www.learnrxjs.io/. 2023 (cit. on p. 66).

[BZA20] A. Boukerche, L. Zheng, O. Alfandi. “Outlier detection: Methods, models, and classification”. In: *ACM Computing Surveys (CSUR)* 53.3 (2020), pp. 1–37 (cit. on p. 18).

[Chy19] Y. Chyung. *Evidence-Based Survey Design: Exclude or Include a Midpoint?* https://www.td.org/insights/evidence-based-survey-design-exclude-or-include-a-midpoint. 2019 (cit. on p. 78).

[DABS22] D. Del Gaudio, B. Ariguib, A. Bartenbach, G. Solakis. “A live context model for semantic reasoning in IoT applications”. In: *2022 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*. IEEE. 2022, pp. 322–327 (cit. on p. 14).

[Dev23a] R. Developers. *RxJS - Documentation*. https://rxjs.dev/guide/overview. 2023 (cit. on pp. 66, 69).

[Dev23b] S. L. Developers. *2.7. Novelty and Outlier Detection*. https://scikit-learn.org/stable/modules/outlier_detection.html. 2023 (cit. on p. 11).

[DH20] D. Del Gaudio, P. Hirmer. “A lightweight messaging engine for decentralized data processing in the internet of things”. In: *SICS Software-Intensive Cyber-Physical Systems* 35.1 (2020), pp. 39–48 (cit. on pp. 13, 14, 33).

[Doc23] Docker Inc. *Dockerfile reference*. https://docs.docker.com/engine/reference/builder/. 2023 (cit. on p. 63).

[DRC20] D. Del Gaudio, M. Reichel, Contributors. *ME Configurator - Source Code*. https://gitlab-as.informatik.uni-stuttgart.de/delgauda/me-configurator. 2020 (cit. on p. 15).

[DRH20] D. Del Gaudio, M. Reichel, P. Hirmer. "A Life Cycle Method for Device Management in Dynamic IoT Environments." In: *IoTBDS*. 2020, pp. 46–56 (cit. on p. 14).

[Eff20] D. Effrosynidis. *Outlier Detection — Theory, Visualizations, and Code*. https://towardsdatascience.com/outlier-detection-theory-visualizations-and-code-a4fd39de540c. 2020 (cit. on pp. 11, 12).

[Eug22] Eugen Paraschiv. *Baeldung - Introduction to Thread Pools in Java*. https://www.baeldung.com/thread-pool-java-and-guava. 2022 (cit. on p. 53).

[FHS+20] A. C. Franco da Silva, P. Hirmer, J. Schneider, S. Ulusal, M. T. Frigo. "MBP: Not just an IoT Platform". In: *2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. 2020, pp. 1–3. DOI: 10.1109/PerComWorkshops48775.2020.9156156 (cit. on pp. 13, 43).

[FOL23] FOLDOC. *shebang from FOLDOC*. https://foldoc.org/shebang. 2023 (cit. on p. 55).

[Gle23] S. Glen. *Sensitivity vs Specificity and Predictive Value*. https://www.statisticshowto.com/probability-and-statistics/statistics-definitions/sensitivity-vs-specificity-statistics/. 2023 (cit. on p. 19).

[Goo23] Google. *CDK Documentation - Angular Material*. https://material.angular.io/cdk/categories. 2023 (cit. on p. 30).

[HHH+22a] S. Han, X. Hu, H. Huang, M. Jiang, Y. Zhao. "ADBench: Anomaly Detection Benchmark". In: *Neural Information Processing Systems (NeurIPS)*. 2022 (cit. on pp. 18, 83).

[HHH+22b] S. Han, X. Hu, H. Huang, M. Jiang, Y. Zhao. *Supplementary Material for ADBench: Anomaly Detection Benchmark*. https://openreview.net/attachment?id=foA_SFQ9zo0&name=supplementary_material. 2022 (cit. on pp. 18, 83).

[HXD03] Z. He, X. Xu, S. Deng. "Discovering cluster-based local outliers". In: *Pattern recognition letters* 24.9-10 (2003), pp. 1641–1650 (cit. on p. 31).

[JHSG+20] J. Jiang, G. Han, L. Shu, M. Guizani, et al. "Outlier detection approaches based on machine learning in the internet-of-things". In: *IEEE Wireless Communications* 27.3 (2020), pp. 53–59 (cit. on pp. 31, 83).

[Job23] Joblib developers. *Joblib: running Python functions as pipeline jobs*. https://joblib.readthedocs.io/en/latest/. 2023 (cit. on p. 58).

[Jon19] Jonathan Cook. *Baeldung - Guide to java.lang.ProcessBuilder API*. https://www.baeldung.com/java-lang-processbuilder-api. 2019 (cit. on p. 53).

[Kar22] F. Karabiber. *Binary Classification*. https://www.learndatasci.com/glossary/binary-classification/. 2022 (cit. on p. 19).

[Kor21] J. Korstanje. *The F1 score*. https://towardsdatascience.com/the-f1-score-bec2bbc38aa6. 2021 (cit. on p. 20).

[KSZ08] H.-P. Kriegel, M. Schubert, A. Zimek. “Angle-based outlier detection in high-dimensional data”. In: *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2008, pp. 444–452 (cit. on p. 31).

[Lin22] Linux man-pages project. *kill(1) — Linux manual page*. https://www.man7.org/linux/man-pages/man1/kill.1.html. Dec. 2022 (cit. on p. 61).

[LSM05] P. J. Leach, R. Salz, M. H. Mealling. *A Universally Unique IDentifier (UUID) URN Namespace*. RFC 4122. July 2005. DOI: 10.17487/RFC4122. URL: https://www.rfc-editor.org/info/rfc4122 (cit. on p. 48).

[LTZ08] F. T. Liu, K. M. Ting, Z.-H. Zhou. “Isolation forest”. In: *2008 eighth ieee international conference on data mining*. IEEE. 2008, pp. 413–422 (cit. on p. 31).

[LTZ12] F. T. Liu, K. M. Ting, Z.-H. Zhou. “Isolation-based anomaly detection”. In: *ACM Transactions on Knowledge Discovery from Data (TKDD)* 6.1 (2012), pp. 1–39 (cit. on p. 31).

[LZW+21] K.-H. Lai, D. Zha, G. Wang, J. Xu, Y. Zhao, D. Kumar, Y. Chen, P. Zumkhawaka, M. Wan, D. Martinez, X. Hu. “TODS: An Automated Time Series Outlier Detection System”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 35.18 (May 2021), pp. 16060–16062 (cit. on pp. 84, 87).

[Mas01] S. Mascheck. *The # magic, details about the shebang/hash-bang mechanism on various Unix flavours*. https://www.in-ulm.de/~mascheck/various/shebang/. 2001 (cit. on p. 55).

[MC22] Microsoft, Contributors. *Pipes and Filters pattern*. https://learn.microsoft.com/en-us/azure/architecture/patterns/pipes-and-filters. 2022 (cit. on p. 32).

[Mon23] MongoDB. *MongoDB Limits and Thresholds*. https://www.mongodb.com/docs/manual/reference/limits/. 2023 (cit. on p. 88).

[Moz23] Mozilla Foundation and Contributors. *The WebSocket API (WebSockets)*. https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API. 2023 (cit. on p. 51).

[Neg20] C. Negus. *Linux Bible - Managing Running Processes*. May 2020. DOI: 10.1002/9781119578956.ch6. URL: http://dx.doi.org/10.1002/9781119578956.ch6 (cit. on p. 61).

[NIS23] NIST. *7.1.6. What are outliers in the data?* https://www.itl.nist.gov/div898/handbook/prc/section1/prc16.htm. 2023 (cit. on p. 11).

[Nya20] Nya. *RxJS Best Practices*. https://dev.to/nyagarcia/rxjs-best-practices-bhb. 2020 (cit. on p. 66).

[Ora14] Oracle. *Java Platform SE 8 - Documentation*. https://docs.oracle.com/javase/8/docs/api/. 2014 (cit. on pp. 53, 55, 61).

[pyo23] pyodide Authors and Contributors. *pyodide Documentation*. https://pyodide.org/en/stable/. 2023 (cit. on p. 47).

[PyP15] PyPy.js Authors and Contributors. *PyPy.js - Website / Documentation*. https://pypyjs.org/. 2015 (cit. on p. 47).

[PyP19] PyPy.js Authors and Contributors. *PyPy.js - Source Code*. https://github.com/pypyjs/pypyjs. 2019 (cit. on p. 47).

[Pyt23] Python Software Foundation and Contributors. *Python 3.10 Documentation*. https://docs.python.org/3.10/. 2023 (cit. on p. 54).

[RRS00] S. Ramaswamy, R. Rastogi, K. Shim. "Efficient algorithms for mining outliers from large data sets". In: *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*. 2000, pp. 427–438 (cit. on pp. 31, 88).

[RVG+18] L. Ruff, R. Vandermeulen, N. Goernitz, L. Deecke, S. A. Siddiqui, A. Binder, E. Müller, M. Kloft. "Deep one-class classification". In: *International conference on machine learning*. PMLR. 2018, pp. 4393–4402 (cit. on p. 31).

[SCSC03] M.-L. Shyu, S.-C. Chen, K. Sarinnapakorn, L. Chang. *A novel anomaly detection scheme based on principal component classifier*. Tech. rep. Miami Univ Coral Gables Fl Dept of Electrical and Computer Engineering, 2003 (cit. on p. 31).

[Sha21] Shawn Shi. *REST API Best Practices — Decouple Long-running Tasks from HTTP Request Processing*. https://medium.com/geekculture/rest-api-best-practices-decouple-long-running-tasks-from-http-request-processing-9fab2921ace8. 2021 (cit. on p. 52).

[SHB14] Z. Shelby, K. Hartke, C. Bormann. *RFC 7252: The constrained application protocol (CoAP)*. 2014 (cit. on p. 14).

[Sur20] A. Suresh. *What is a confusion matrix?* https://medium.com/analytics-vidhya/what-is-a-confusion-matrix-d1c0f8feda5. 2020 (cit. on p. 19).

[TCFC02] J. Tang, Z. Chen, A. W.-C. Fu, D. W. Cheung. "Enhancing effectiveness of outlier detections for low density patterns". In: *Pacific-Asia conference on knowledge discovery and data mining*. Springer. 2002, pp. 535–548 (cit. on p. 31).

[VMW23] VMWare Tanzu and Contributors. *Spring Framework Documentation*. https://docs.spring.io/spring-framework/docs/current/reference/html/index.html. 2023 (cit. on pp. 51, 62).

[Wei22] E. W. Weisstein. *Lissajous Curve. From MathWorld–A Wolfram Web Resource*. https://mathworld.wolfram.com/LissajousCurve.html. 2022 (cit. on p. 22).

[Woo22] T. Wood. *DeepAI - F-Score*. https://deepai.org/machine-learning-glossary-and-terms/f-score. 2022 (cit. on p. 20).

[Woz22] D. Wozniak. *24 Angular Best Practices You Shouldn't Code Without*. https://massivepixel.io/blog/angular-best-practices/. 2022 (cit. on p. 65).

[XKC19] Z. Xu, D. Kakde, A. Chaudhuri. "Automatic hyperparameter tuning method for local outlier factor, with applications to anomaly detection". In: *2019 IEEE International Conference on Big Data (Big Data)*. IEEE. 2019, pp. 4201–4207 (cit. on p. 83).

[ZH18] Y. Zhao, M. K. Hryniewicki. "XGBOD: improving supervised outlier detection with unsupervised representation learning". In: *2018 International Joint Conference on Neural Networks (IJCNN)*. IEEE. 2018, pp. 1–8 (cit. on p. 31).

[Zha23] Zhao, Yue and Nasrullah, Zain and Li, Zheng and pyOD Authors. *pyOD Documentation*. https://pyod.readthedocs.io/en/latest/index.html. 2023 (cit. on pp. 22, 47, 58, 83).

[ZNL19] Y. Zhao, Z. Nasrullah, Z. Li. "PyOD: A Python Toolbox for Scalable Outlier Detection". In: *Journal of Machine Learning Research* 20.96 (2019), pp. 1–7. URL: http://jmlr.org/papers/v20/19-011.html (cit. on pp. 22, 47, 83).

[ZRA20] Y. Zhao, R. Rossi, L. Akoglu. “Automating Outlier Detection via Meta-Learning”. In: *arXiv preprint arXiv:2009.10606* (2020) (cit. on pp. 83, 87).

All links were last followed on March 1, 2023.

A Appendix

A.1 Data Collection for temperature measurements on a smart plug

The overall flow of the data collection for the temperature measurements in the real world dataset works as follows:

1. The measurement is transmitted to the base station (A Fritz!Box 7590) wirelessly[1].
2. The base station allows the reading of the current measurement by using an API.
3. A tool called `fritzbox_smarthome_exporter`[2] exposes the measurements from the base station as a Prometheus[3] compatible endpoint.
4. The endpoint is crawled in regular intervals of 15 seconds by `telegraf`[4] that stores the collected measurements in a PostgreSQL[5].

The database can then be queried to extract the data as a time series or a `csv` file.

[1]Exact technical details are unknown and do not matter for the data collection

[2]`https://github.com/jayme-github/fritzbox_smarthome_exporter`

[3]`https://prometheus.io/`

[4]`https://docs.influxdata.com/telegraf/v1.25/`

[5]`https://www.postgresql.org/`

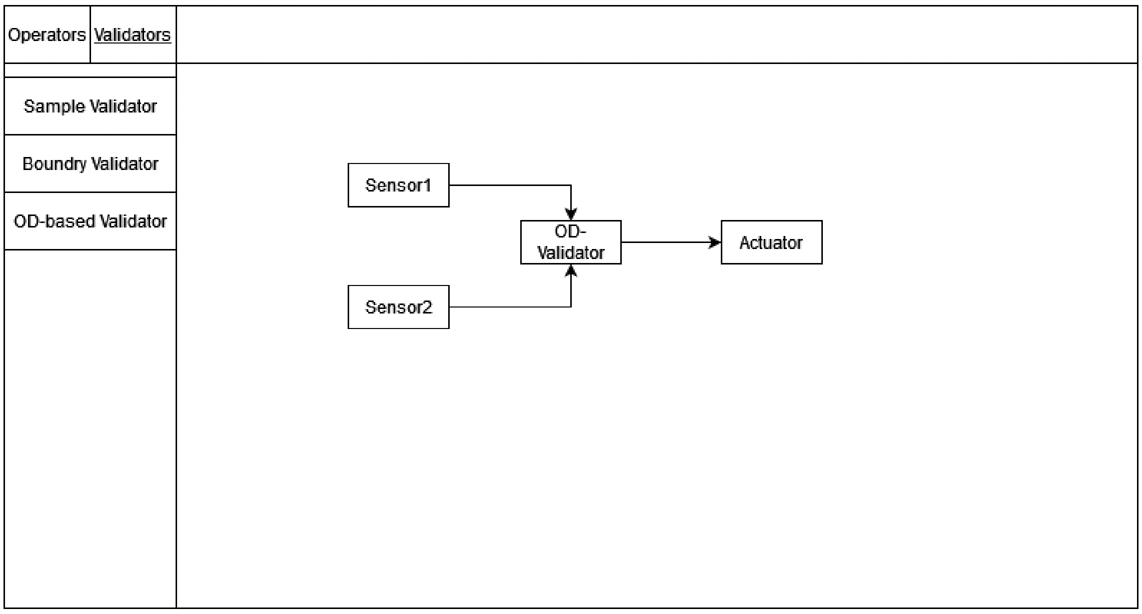


Figure A.1: Wireframe of the IoT Application Modelling Tool, modelling a dual input validator with one actuator

A.2 Figures

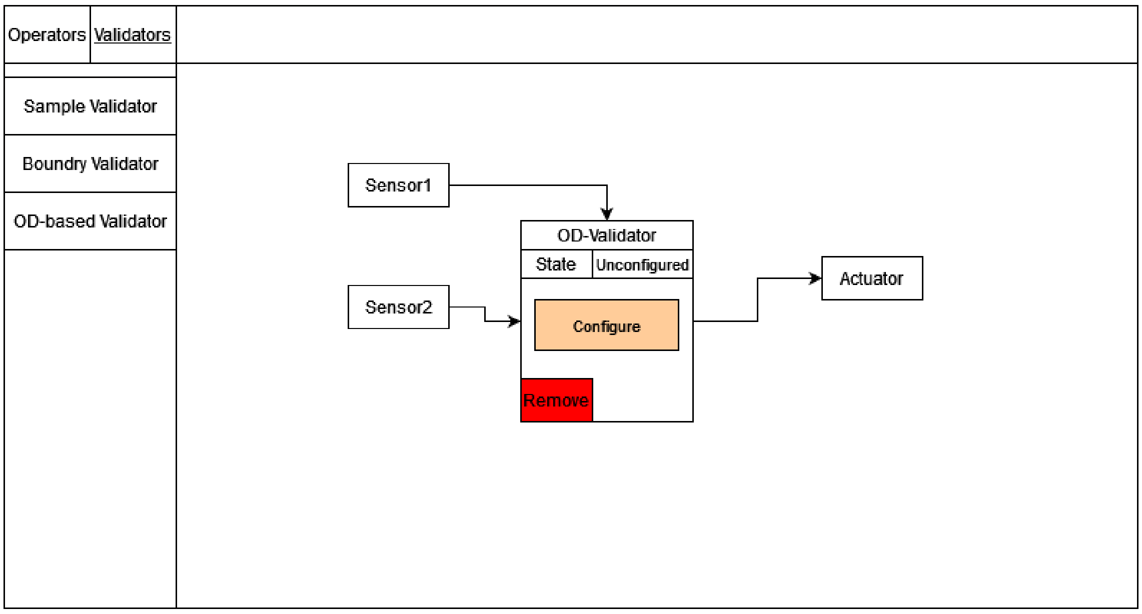
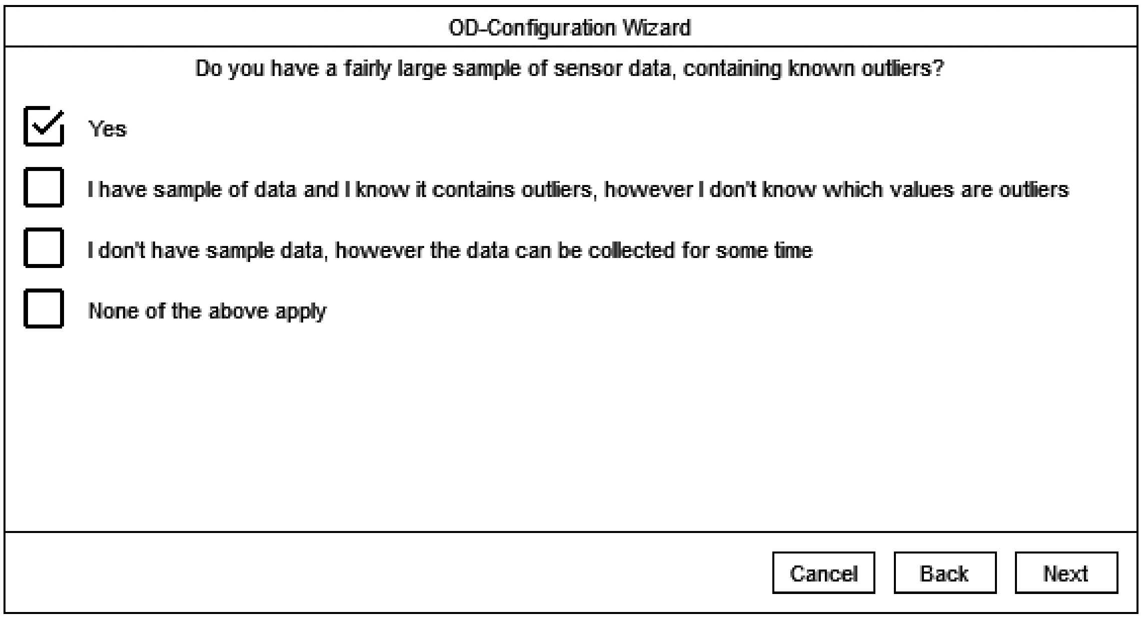


Figure A.2: Wireframe of the IoT Application Modelling Tool, modelling a dual input validator with one actuator, with an expanded unconfigured validator


OD-Configuration Wizard

Do you have a fairly large sample of sensor data, containing known outliers?

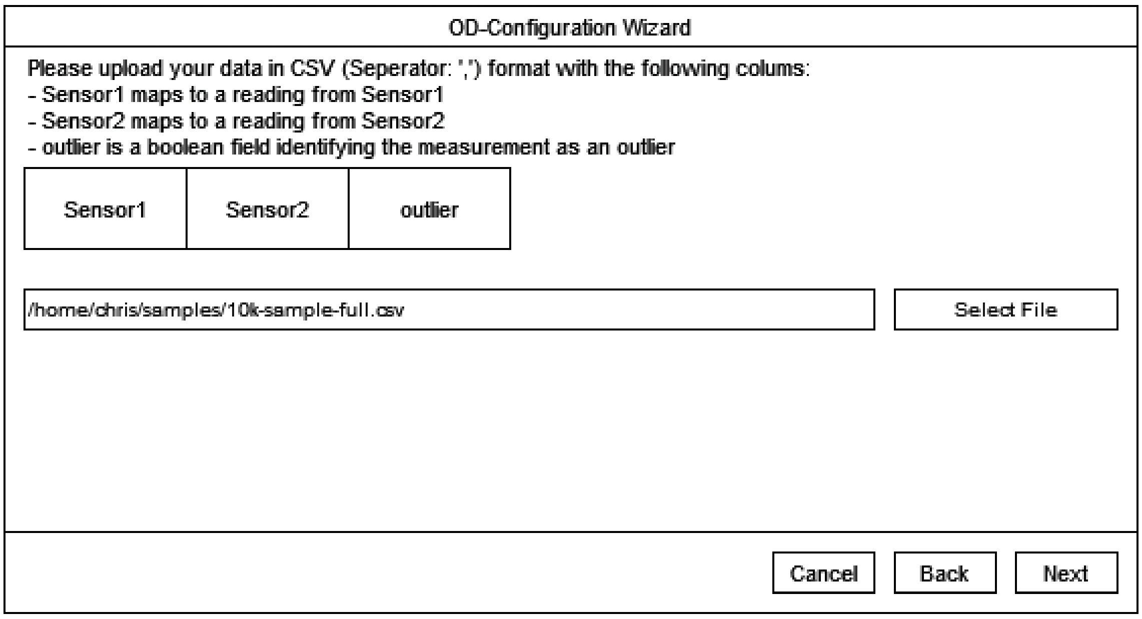
[✓ Yes]

I have sample of data and I know it contains outliers, however I don't know which values are outliers

I don't have sample data, however the data can be collected for some time

None of the above apply

Figure A.3: Wireframe of the first, introductory, step of the wizard



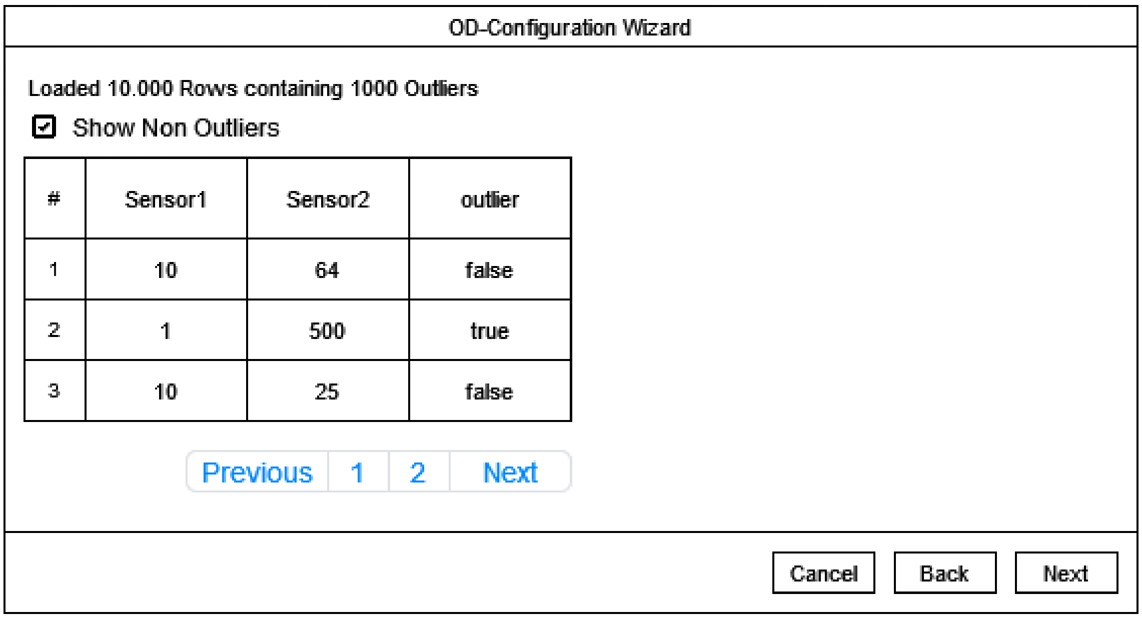
OD-Configuration Wizard

Please upload your data in CSV (Seperator: ',') format with the following colums:
- Sensor1 maps to a reading from Sensor1
- Sensor2 maps to a reading from Sensor2
- outlier is a boolean field identifying the measurement as an outlier

Sensor1	Sensor2	outlier

/home/chris/samples/10k-sample-full.csv

Figure A.4: Wireframe of the Data Upload step in the wizard



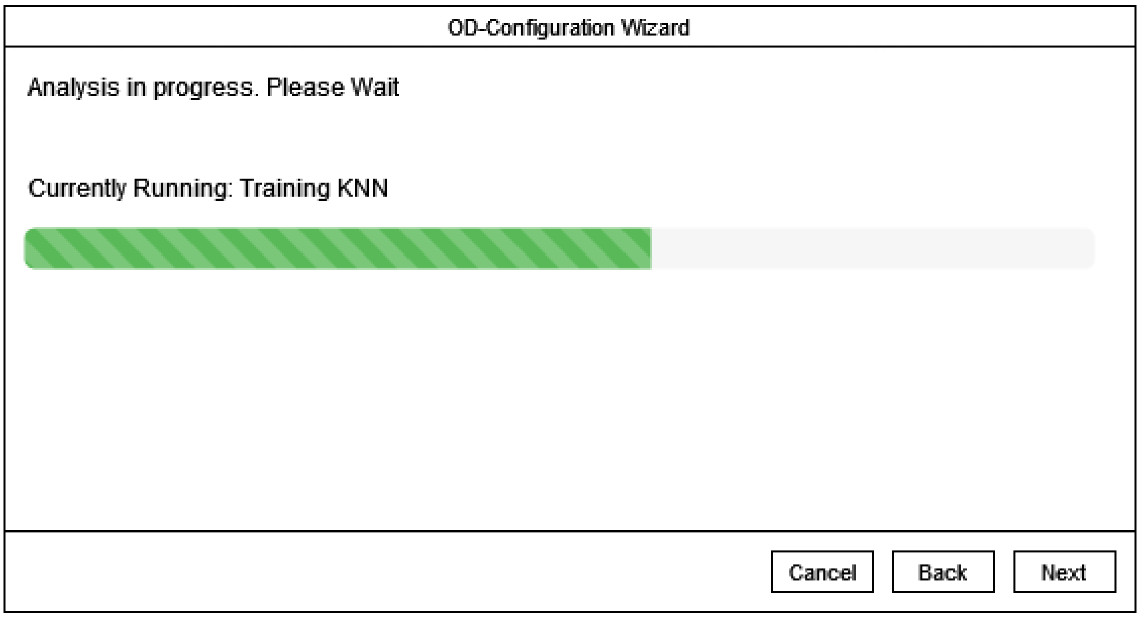
OD-Configuration Wizard

Loaded 10.000 Rows containing 1000 Outliers

☑ Show Non Outliers

#	Sensor1	Sensor2	outlier
1	10	64	false
2	1	500	true
3	10	25	false

Figure A.5: Wireframe of the data review step in the wizard


OD-Configuration Wizard

Analysis in progress. Please Wait

Currently Running: Training KNN

Figure A.6: Wireframe of the job running (progress display) step in the wizard

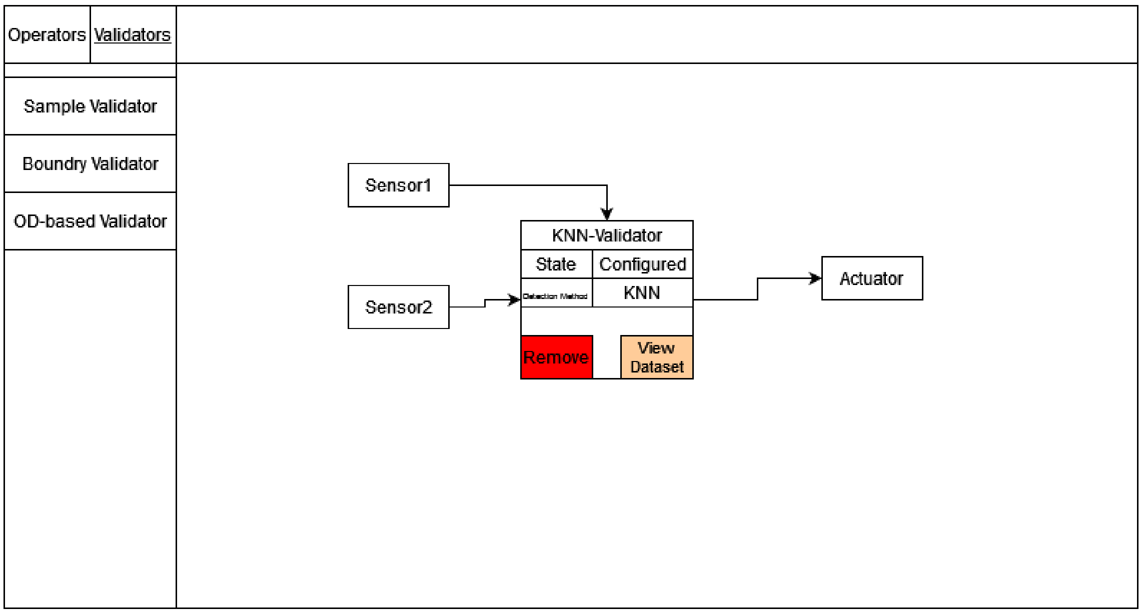
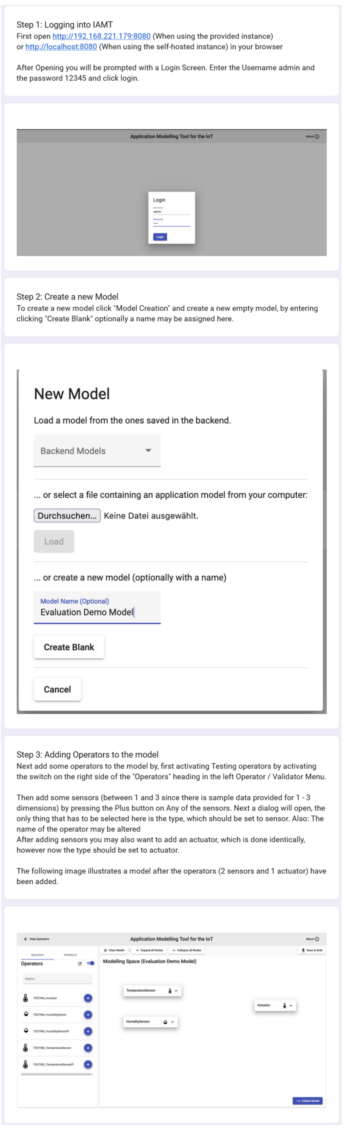


Figure A.7: Wireframe of the configured validator



Step 1: Logging into IAMT
First open http://192.168.221.179:8080 (When using the provided instance)
or http://localhost:8080 (When using the self-hosted instance) in your browser

After Opening you will be prompted with a Login Screen. Enter the Username admin and the password 12345 and click login.

Step 2: Create a new Model
To create a new model click "Model Creation" and create a new empty model, by entering clicking "Create Blank" optionally a name may be assigned here.

New Model

Load a model from the ones saved in the backend.

... or select a file containing an application model from your computer:

... or create a new model (optionally with a name)

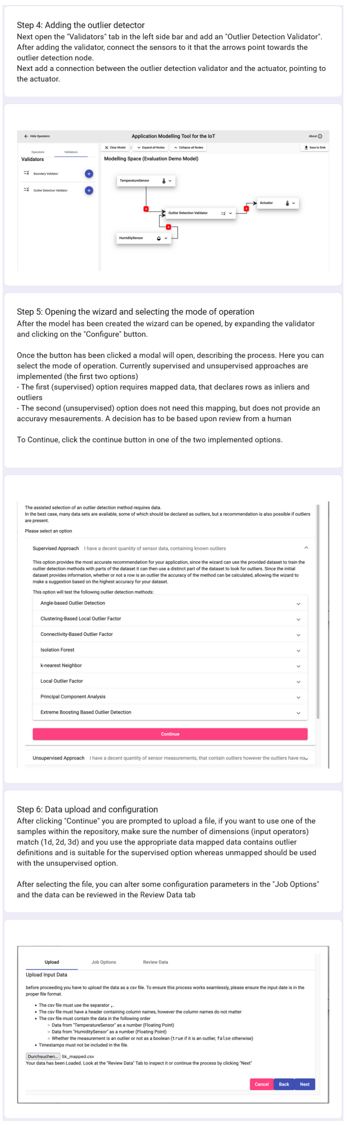
Model Name (Optional): Evaluation Demo Model

Step 3: Adding Operators to the model
Next add some operators to the model by, first activating Testing operators by activating the switch on the right side of the "Operators" heading in the left Operator / Validator Menu.

Then add some sensors (between 1 and 3 since there is sample data provided for 1 - 3 dimensions) by pressing the Plus button on Any of the sensors. Next a dialog will open, the only thing that has to be selected here is the type, which should be set to sensor. Also: The name of the operator may be altered
After adding sensors you may also want to add an actuator, which is done identically, however now the type should be set to actuator.

The following image illustrates a model after the operators (2 sensors and 1 actuator) have been added.

(a) Steps 1 to 3



Step 4: Adding the outlier detector
Next open the "Validators" tab in the left side bar and add an "Outlier Detection Validator". After adding the validator, connect the sensors to it that the arrows point towards the outlier detection node.
Next add a connection between the outlier detection validator and the actuator, pointing to the actuator.

Step 5: Opening the wizard and selecting the mode of operation
After the model has been created the wizard can be opened, by expanding the validator and clicking on the "Configure" button.

Once the button has been clicked a modal will open, describing the process. Here you can select the mode of operation. Currently supervised and unsupervised approaches are implemented (the first two options)
- The first (supervised) option requires mapped data, that declares rows as inliers and outliers
- The second (unsupervised) option does not need this mapping, but does not provide an accuravy mesaurements. A decision has to be based upon review from a human

To Continue, click the continue button in one of the two implemented options.

The assisted selection of an outlier detection method requires data.
In the best case, many data sets are available, some of which should be declared as outliers, but a recommendation is also possible if outliers are present.

Please select an option

Supervised Approach I have a decent quantity of sensor data, containing known outliers

This option provides the most accurate recommendation for your application, since the wizard can use the provided dataset to train the outlier detection methods with parts of the dataset it can then use a distinct part of the dataset to look for outliers. Since the initial dataset provides information, whether or not a row is an outlier the accuracy of the method can be calculated, allowing the wizard to make a suggestion based on the highest accuracy for your dataset.

This option will test the following outlier detection methods:

- Angle-based Outlier Detection
- Clustering-Based Local Outlier Factor
- Connectivity-Based Outlier Factor
- Isolation Forest
- k-nearest Neighbor
- Local Outlier Factor
- Principal Component Analysis
- Extreme Boosting Based Outlier Detection

Unsupervised Approach I have a decent quantity of sensor measurements, that contain outliers however the outliers have no

Step 6: Data upload and configuration
After clicking "Continue" you are prompted to upload a file, if you want to use one of the samples within the repository, make sure the number of dimensions (input operators) match (1d, 2d, 3d) and you use the appropriate data mapped data contains outlier definitions and is suitable for the supervised option whereas unmapped should be used with the unsupervised option.

After selecting the file, you can alter some configuration parameters in the "Job Options" and the data can be reviewed in the Review Data tab

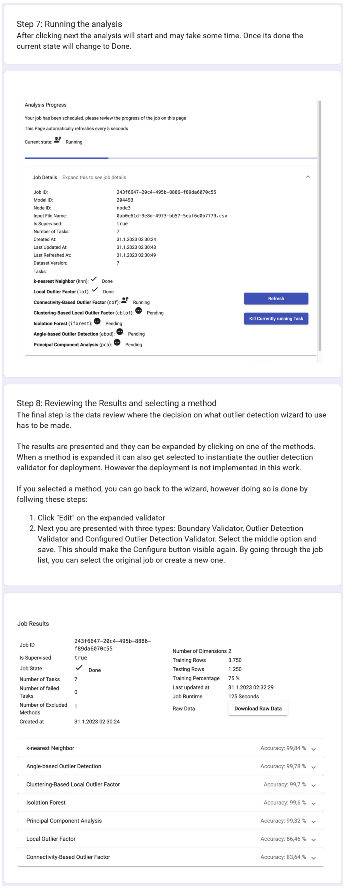
Upload Input Data

before proceeding you have to upload the data as a csv file. To ensure this process works seamlessly, please ensure the input date is in the proper file format.

- The csv file must use the separator `,`.
- The csv file must have a header containing column names, however the column names do not matter
- The csv file must contain the data in the following order
  - Data from "TemperatureSensor" as a number (Floating Point)
  - Data from "HumiditySensor" as a number (Floating Point)
  - Whether the measurement is an outlier or not as a boolean (`true` if it is an outlier, `false` otherwise)
- Timestamps must not be included in the file.

Durchsuchen... 5k_mapped.csv
Your data has been Loaded. Look at the "Review Data" Tab to inspect it or continue the process by clicking "Next"

(b) Steps 4 to 6



Step 7: Running the analysis
After clicking next the analysis will start and may take some time. Once its done the current state will change to Done.

Analysis Progress

Your job has been scheduled, please review the progress of the job on this page

This Page automatically refreshes every 5 seconds

Current state: Running

Job Details Expand this to see job details

Job ID:	243f6647-20c4-495b-8886-f89da6070c55
Model ID:	204493
Node ID:	node3
Input File Name:	0ab0e61d-9e8d-4973-bb57-5eaf6d0b7779.csv
Is Supervised:	true
Number of Tasks:	7
Created At:	31.1.2023 02:30:24
Last Updated At:	31.1.2023 02:30:43
Last Refreshed At:	31.1.2023 02:30:49
Dataset Version:	7

Tasks:
k-nearest Neighbor (knn): ✓ Done
Local Outlier Factor (lof): ✓ Done
Connectivity-Based Outlier Factor (cof): Running
Clustering-Based Local Outlier Factor (cblof): Pending
Isolation Forest (iforest): Pending
Angle-based Outlier Detection (abod): Pending
Principal Component Analysis (pca): Pending

Step 8: Reviewing the Results and selecting a method
The final step is the data review where the decision on what outlier detection wizard to use has to be made.

The results are presented and they can be expanded by clicking on one of the methods. When a method is expanded it can also get selected to instantiate the outlier detection validator for deployment. However the deployment is not implemented in this work.

If you selected a method, you can go back to the wizard, however doing so is done by follwing these steps:

1. Click "Edit" on the expanded validator
2. Next you are presented with three types: Boundary Validator, Outlier Detection Validator and Configured Outlier Detection Validator. Select the middle option and save. This should make the Configure button visible again. By going through the job list, you can select the original job or create a new one.

Job Results

Job ID	243f6647-20c4-495b-8886-f89da6070c55	Number of Dimensions	2
Is Supervised	true	Training Rows	3.750
Job State	✓ Done	Testing Rows	1.250
Number of Tasks	7	Training Percentage	75 %
Number of failed Tasks	0	Last updated at	31.1.2023 02:32:29
Number of Excluded Methods	1	Job Runtime	125 Seconds
Created at	31.1.2023 02:30:24	Raw Data	Download Raw Data

k-nearest Neighbor	Accuracy: 99,84 %
Angle-based Outlier Detection	Accuracy: 99,78 %
Clustering-Based Local Outlier Factor	Accuracy: 99,7 %
Isolation Forest	Accuracy: 99,6 %
Principal Component Analysis	Accuracy: 99,32 %
Local Outlier Factor	Accuracy: 86,46 %
Connectivity-Based Outlier Factor	Accuracy: 83,64 %

(c) Steps 7 to 9

Figure A.8: The tutorial shown as part of the questionaire

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature