UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR IT-SICHERHEIT

Context Aware Anomaly Detection in Industrial Control Systems via Machine Learning

Kontextbewusste Anomaliedetektion in industriellen Steuerungssystemen mittels maschinellem Lernen

Masterarbeit

im Rahmen des Studiengangs
Informatik
der Universität zu Lübeck

Vorgelegt von
Moritz Welberg

Ausgegeben und betreut von
Prof. Dr.-Ing. Thomas Eisenbarth

Mit Unterstützung von
Dr. Simon Walz (cbb software GmbH)

und
M.Sc. Claudius Pott

Lübeck, den 18. November 2021

Abstract

There are old and unsupported (legacy) devices in industrial control systems. Erroneous devices might endanger security goals of confidentiality, integrity and availability of these systems' data. Therefore the emitted data of such devices needs to be monitored since it may provide information on misbehaving or even maliciously behaving devices and overall systems. Monitoring data to detect anomalies indicative of unwanted behaviour and selecting useful detection methods often requires costly expert knowledge. Automating the selection of useful anomaly detection methods for univariate time series data could be a viable approach to reduce the required expert knowledge and subsequently reduce the cost of such detection systems for small and medium sized companies. We designed, implemented and evaluated an evaluation based approach in which we inject artificial anomalies into training datasets to select a useful anomaly detection method for a group of similar data sources. Clustering similar data sources to assign one anomaly detection method to many data sources is faster than a brute force approach. Our evaluations show that the real performance of a selected method is strongly dependent on the type and number of anomalies injected during the method evaluation. We found that randomly injected artificial point anomalies enable our system to select the best detection method for a data source prone to random point anomalies. Artificially and randomly repeated subsequences do not lead to a usable ranking of detection methods for these data sources. The approach of clustering similar data sources and assigning detection methods group wise is promising. Injecting anomalies into a dataset for method selection without knowing typical or possible forms of anomalies of such dataset may lead to falsely selected methods and is not recommended without further analysis of possible anomaly structures.

Zusammenfassung

Es gibt in industriellen Steuerungssystemen oft veraltete und nicht weiter unterstützte (legacy) Geräte. Sind diese Geräte fehlerhaft, können sie mitunter eine Gefährdung der IT-Sicherheitsziele der Geheimhaltung, Integrität und Verfügbarkeit von Systemdaten darstellen. Daher sollten die erzeugten Daten solcher Geräte überwacht werden, da diese Hinweise auf ungewünschtes oder gar bösartiges Verhalten der Geräte und entsprechenden Gesamtsysteme geben können. Solche Überwachung auf Anomalien und die Auswahl passender Detektionsmethoden benötigt teilweise sehr teures Expertenwissen. Die Automatisierung der Auswahl von passenden Anomaliedetektionsmethoden für eindimensionale Zeitsequenzdaten scheint ein vielversprechender Weg zu sein um benötigtes Expertenwissen und daraus resultierende Kosten für kleine und mittelständische Unternehmen zu reduzieren. Wir haben ein System entworfen, implementiert und evaluiert, welches in einem evaluationsbasiertem Ansatz künstliche Anomalien in Trainingsdatensätze einfügt um passende Anomaliedetektionsmethoden für eine Gruppe von Datenquellen auszuwählen. Die Clusterbildung von ähnlichen Datenquellen zur gleichzeitigen Zuweisung von Detektionsmethoden für viele Datenquellen ist schneller als ein Brute-Force Ansatz, welcher jede Methode für jede Quelle einzeln überprüft. Unsere Evaluationen zeigen, dass die tatsächliche Güte einer ausgewählten Detektionsmethode stark von der Zahl und Art der in den Trainingsdatensatz eingefügten Anomalien abhängt. Zufällig eingefügte punktuelle Anomalien erlauben unserem System die beste Detektionsmethode für Datenquellen zu wählen, die anfällig für zufällige, punktuelle Anomalien sind, während zufällig wiederholte Teilsequenzen als eingefügte Anomalien nicht zur sinnvollen Bewertung von Methoden für diese Datenquellen führen. Der Ansatz der Clusterbildung von ähnlichen Datenquellen und der entsprechenden gleichzeitigen Zuweisung von Detektionsmethoden ist vielversprechend. Das Einfügen von Anomalien in einen Trainingsdatensatz ohne vorherige Kenntnis von realistischen Anomalieszenarien der entsprechenden Datenquelle ist ohne eine Analyse möglicher Anomaliestrukturen nicht empfehlenswert.

Erklärung

Ich versichere an Eides statt, die vorliegende Arbeit selbstständig und nur unter Benutzung der angegebenen Hilfsmittel angefertigt zu haben.

Lübeck, der 18. November 2021

Acknowledgements

I want to thank Prof. Dr. Thomas Eisenbarth who accepted and discussed my initial idea for this thesis and posed critical questions early on. Dr. Simon Walz was a great help in developing the concept of the prototype implementation and removing organizational barriers for me. Claudius Pott M.Sc. and Dr. Madline Kniebusch were always ready to answer my questions. They each read an commented on multiple iterations of the thesis text and gave necessary advice on the prototype implementation in numerous and extensive discussions. I also want to thank the staff of the University of Lübeck Institute for IT Security and the cbb software GmbH for countless supportive interactions.

Contents

1 Introduction

Legacy devices in Industrial Control Systems (ICS) connected to the internet are a security risk. Replacing those devices can be a huge investment without any immediate positive effect on productivity and profitability.
Keeping the legacy devices while monitoring their behaviour to detect attacks on endpoints, lateral movement of attackers or erroneous value/parameter divergences appears to be an economically feasible strategy to mitigate the stated risks until 'security by design'-standards and -devices for ICS have been widely adopted.
Most research is focused on anomalies in telecommunication network traffic (probably because many ICS Intrusion Detection Systems (IDS) are derived from enterprise IDS) where deep packet inspection is in many cases unwanted, unnecessary or even illegal [SFT+20, MC14].
Such constrains do not hold for machine-to-machine communications in ICS Networks, so deep packet inspection for an anomaly detection system (ADS) based on process data combined with traffic metadata could prove to be much more effective, than analysing traffic metadata on its own.
Since an ADS typically requires an in in-depth analysis of the observable data and in many cases domain specific knowledge, the development and continuous maintenance of such systems is usually conducted by experts, which pose a financial barrier to ADS access for smaller enterprises [AFS17].
The general goal of this thesis is to design and implement a prototype application for anomaly detection in ICSs with a minimum of required user interaction. Ideally a finished product, based on this prototype would be installed in a customers ICS environment and trained with benign time series data from sensors and processes. Without any user interaction the system can select the most fitting anomaly detection method for each time series data source and apply this method continuously on newly arriving streams of data. Towards reaching this goal we employ methods of unsupervised machine learning in a meta-learning framework used to select the best anomaly detection method suited for some input data. Our focus lies on the following research questions:

- How can we evaluate anomaly detection methods on unlabelled data? A major problem when evaluating anomaly detection methods is that there are few labelled datasets and their usability for the problem at hand may vary drastically [CBK09]. Injecting anomalies into some unlabelled dataset according to some simple heuristic,

appears a useful approach to handle this problem [SG19]. We explore and evaluate different simple heuristics and their effects on the performance of anomaly detection methods.

- How can we select anomaly detection methods for a large number of data sources? Choosing the best anomaly detection method for each single data source can quickly create a huge computational cost when many data sources are involved. We explore an approach where we do not need to evaluate each method for every single data-source on its own, similar to [ZRA20].

- How can we efficiently measure similarity of input datasets? Working in a real time setting, the computational cost of comparing incoming data to existing data needs to be minimized. We evaluate two different clustering mechanisms for our use-case.

Chapter 2 after this introduction provides an overview of publications related to the field. The succeeding Chapter 3 contains explanations of the methods and algorithms used in our prototype as well as a stringent reasoning why these methods were chosen. It follows Chapter 4 on the implementation, where the general architecture as well as some particular modules are highlighted in detailed descriptions. Chapter 5 is used to illustrate the prototype evaluation process and its subsequent results. In the last Chapter 6 we draw conclusions from the evaluation and discuss some possible improvements as well as future research opportunities.

2 Related Work

To keep the ADS adaptive and to reduce the amount of expert knowledge needed to maintain the ADS, a system architecture should enable the user to configure the system and extract valuable insights from the available data without the need of an extensive data analysis skill set. Therefore not only the task of detecting anomalous samples, but the selection of a well performing detection method should be automated too. A sophisticated architecture for general machine learning systems fulfilling this requirement is proposed by Kadlec et al. in their work on adaptive online prediction models [KG09].
The strategy of first classifying the received data and selecting a detection model based on this classification is used by Dreger et al. to identify malicious packages in enterprise networks. For each connection, their system first identifies potential network protocols in use, based on a dynamic processing path. The system then verifies the decision with an option to decline further connection processing and extracts higher-level semantics for a signature based analysis [DFM+06].
Zhao et al. propose the METAOD *meta-learner* for unsupervised outlier detection model selection. The attributes (features) of every new dataset are compared to previously analysed datasets. Once some sufficiently similar datasets are found, they use these similar datasets to predict the anomaly detection performance of some pre-trained classifiers for the new dataset. Based on this prediction they return the classifier with the best predicted performance for outlier detection on the new dataset [ZRA20].
Wressnegger et al. recently introduced a method for content based anomaly detection on binary protocol data using n-grams. By statistical analysis and clustering, they build a sparse feature matrix characterizing industrial control system protocols in an abstract way. These abstractions are then used to inject attack payloads into arbitrary protocol traffic and to detect such attacks in arbitrary protocols, with great success in comparison to other n-gram and content based anomaly detection methods [WR].
A different approach to classify (in this case string based) process data is used by Marschalek et al., when they build a distance matrix based on behavioural profiles, characterized by Markov chains. The generated distance matrices are used to find strings, which are similar or equal (small or zero distance) and assign them to the same cluster [MLS18].
These and similar studies [CBK09] show, that clustering, based on a distance metric can be considered a viable technique for unsupervised sequential data classification and we

use it as such.
Regarding the detection of anomalies, a recent study by Anton et al. compares three methods of anomaly detection on Modbus Transmission Control Protocol (TCP) data: *Matrix Profiles* (MP), Seasonal Autoregressive Integrated Moving Average (SARIMA) and Long Short-Term Memory (LSTM) neural networks. Matrix Profiles and SARIMA are statistical methods with high accuracy and little to no prior configuration. *LSTM* is a specific kind of recurrent neural network, which is able to compete with the statistical approaches, but needs more data and more configuration than other methods. Each of the three algorithms and respective experiments underline that sequence and time information plays an essential role in accurately detecting anomalies or intrusions in industrial process data [DAFS19]. Especially Matrix profiles and *SARIMA* are promising candidates for unsupervised anomaly detection. However unsupervised deep learning methods such as autoencoders (AE) and *LSTM-AE* appear to have potential in domains other than Modbus TCP data [CC19]. We use the *Matrix Profile* and a *LSTM* based methods to detect anomalies.
Flosbach et al. present an architecture of a prototype implementation for a context aware IDS where traffic processing is separated from event evaluation. The context is created by modelling a topology representation and a system state such that changes in network topology and potentially harmful system states can be detected [FCR19]. We will use a similar approach and strictly separate traffic processing from event evaluation by using an already existing middleware that lets us retrieve process data using a REST (Representational State Transfer) API (Application Programming Interface).
One drawback in most of the evaluated implementations of ADS is the incomparability caused by insufficient performance metrics [MC14]. Additionally to detection-rate (F-Score), false positive rate (FPR) and false negative rate (FNR), the detection latency (e.g. time delta of attack time and detection time) is an important metric to evaluate anomaly detection systems and should be considered when comparing different techniques and methodologies but is often not evaluated.

2.1 Intrusion Detection by Anomaly Detection

One application domain of anomaly detection is the detection of unauthorized intrusions into computer networks. The underlying hypothesis is that any (unauthorized) attempt of tampering with the general security goals of confidentiality, integrity and availability, must result in some sort of anomalous system behaviour. When monitoring the relevant data sources, applying established anomaly detection methods should result in indica-

tions of intrusions.
Current IDS use many different approaches to distinguish benign from anomalous behaviour. These methods range from simple whitelisting over signature comparisons and behaviour rule specifications to sophisticated machine learning algorithms [Sch19, KUF+20, AMK+20]. Our approach employs methods of unsupervised machine learning that have been used successfully in this domain in previous publications [AKFS18, DAFS19]. Contrasting the already widely used approach of supervised anomaly and intrusion detection in Supervisory Control and Data Acquisition (SCADA) systems [SFT+20, KUF+20].
We evaluate our system using anomaly detection benchmark datasets [ALPA17]. The insights these evaluations provide are twofold. We gain information on the general performance of our prototype system and its single components and are also able to derive implications on the feasibility of our approach in real world scenarios.

3 Methods

In this chapter we describe the details of the methods used to implement the system prototype. The general approach is to first elaborate use cases of the method and form an intuition of the whole concept. Then we name the core ideas and formalize them with definitions and algorithms where needed.
In accordance with the three questions stated in Chapter 1, we first discuss how to prepare time series for anomaly detection, what anomalies are and the methods to find them. Then we explain how characteristics of time series can be used to build clusters of similar time series. In the end, we describe the concept of Meta-Learning and which evaluation parameters are used to compare different anomaly detection methods.

3.1 Time Series Analysis

As we aim to find similarities of different time series datasets we first need to identify and extract characteristic attributes of time series. Time series analysis is used for such extraction and is usually part of a larger data mining pipeline consisting of data collection, cleaning, processing, analysing and interpretation. As most data mining tasks, time series analysis is a valuable tool for many different domains to create condensed data summaries allowing for recommendations on current or future actions [Agg15]. Application domains of time series analysis include, but are not limited to, climate and weather research, health monitoring, manufacturing and many more.

3.1.1 Data Collection

Datasets can be aggregated from many different data sources. In the context of ICS there often exist hundreds of data sources, differing in type and functionality. This abundance of data is often captured by systems and databases implemented specifically for the task of collecting the raw data output from data sources and providing a uniform interface to interact with this data. We assume such middleware exists and provides REST API endpoints for us to fetch the data samples from. Since our system should also be able to learn from static, historical data we also allow CSV (Comma Separated Values) datasets to be handed to the system.

3.1.2 Preprocessing

Preprocessing has to be conducted even before the first method of time series analysis might be applied. This step is necessary to convert raw data into a usable and clean dataset in a form which can be processed by further methods. Although there are many generally applicable methods of preprocessing, this step is often highly specialized for each individual source of data. The following three preprocessing techniques are more general applicable examples used in our prototype implementation.

Reduction of Dimensions Many time series datasets contain multiple values for each time step such as a timestamp, the raw value, some moving average or other meta information. To reduce such data to a single dimension one might sort the data by its timestamp and then discard meta information, leaving only the raw value to work with.

Subsampling Depending on the resolution of the recorded data, one might be able to discard every other sample without losing much of the information given by the whole time series. This technique can be used to eliminate redundant data values and thereby accelerate further processing.

Scaling Many machine learning methods require their input data to be scaled. Else they might not converge or otherwise deliver useful results. Common scaling techniques are *Normalization*, where the data is fitted in between a minimum (usually 0) and a maximum (usually 1) value, and *Standardization*, where the mean of the data is shifted to 0 and the variance to 1 (see Appendix 1) [PVG+11]. We use both techniques in different contexts of our prototype implementation.

3.1.3 Features of Time Series

To compare different time series we need to take a look at characteristics or features defining the time series.

Periodicity, Trend and Noise Three often used characteristics of time series data are periodicity, trend and noise [PBDM20].
The operations needed to extract information on periodicity are complex and computationally expensive. Simpler and faster methods to reach the same goal (extracting a vector of characteristic features) are needed for our use case. We explored periodicity as a single feature contributing to the comparison of different time series but came to the conclusion

that many more features are necessary to accomplish this task. A single binary feature (is periodic/is not periodic) lacks the depth to characterize a whole dataset.

Statistical Features In practice, a feature vector describing some data point should consist of many more dimensions. A python library to extract high dimensional feature vectors from time series is *tsfresh* [CBNKL18].
Although using this library for feature extraction on streaming data is not recommended, since it is not explicitly implemented, we found it sufficient for our proof of concept implementation.
Our selected set of statistical features provided by *tsfresh* contains computations of the median, mean, standard deviation, variance and more (see Appendix Listing 1) of any given time series. All of these have been proven useful in characterizing time series datasets and provide a reasonable initial configuration for a prototype implementation [RSG+14, CBNKL18, ZRA20].

3.2 Anomaly Detection

Collecting meta information on some given time series is just one task we need to manage. Another task is to detect anomalies in these time series. Anomaly detection is the action of finding patterns or single instances in data, which deviate from the expected data. These expectations are mostly defined by specifications or previously observed data.
The tasks where anomaly detection is applied span a broad range from fraud detection in banking transactions, over analysis of malfunctions in hardware to intrusion detection in computer networks. Where there is a massive amount of data, occurrence of anomalies is highly probable. Anomaly detection is used to find these anomalies and helps to identify the cause of the anomalous data [CBK09].

3.2.1 Types of Anomalies

In [CBK09] Chandola et al. distinguish three general types of anomalies, which are independent from any application domain:

- **Point Anomalies** are single samples which deviate from the normal range of samples. An example of multiple point anomalies is given in Figure 3.1. It shows a synthetic dataset of simulated daily temperature measurements. Five values are far above the usual range of measured temperatures and one value is significantly lower than expected. In this case this expectation is based on the density function of the given dataset as described in Section 3.4.1.

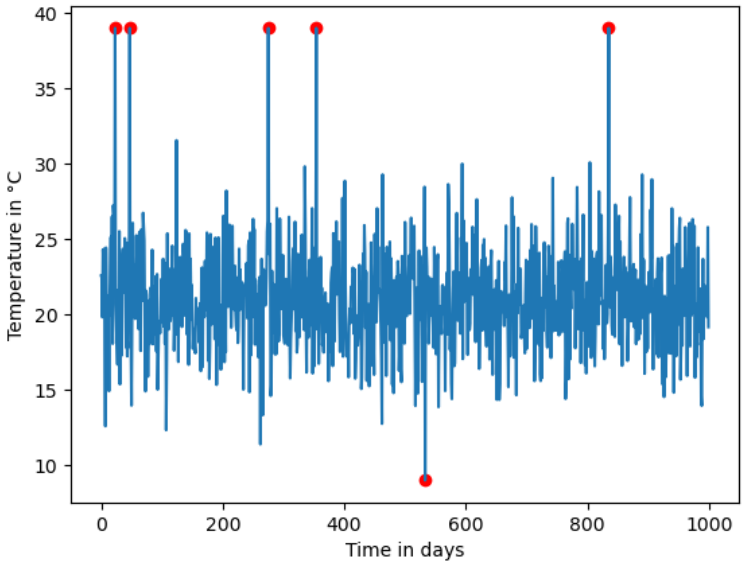

Figure 3.1: Synthetic temperature data containing 6 anomalous samples.

- **Contextual Anomalies** occur when a sample is only considered anomalous if it is present in a specific context. Contextual attributes of a sample are such attributes that define the neighbourhood (e.g. distance in time or space) to other samples. The timestamp of a sample in a time series would enable the observer to compare this sample to other samples closely before and after this sample, making time the contextual attribute. The value (e.g. temperature) of such sample would constitute a behavioural attribute that describes the behaviour of the data source at a given time step. An example is given in Figure 3.2. There are four samples in this time series that would be ordinary at other time steps but are not within their current context.

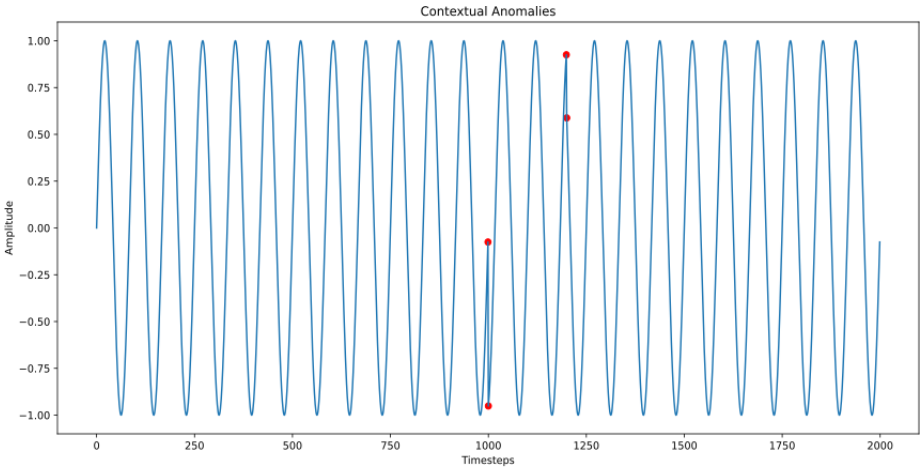

Figure 3.2: Samples well within the range of possible values but at unexpected time steps make contextual anomalies.

- **Collective Anomalies** are collections of samples, which differ from most other collections of samples in a dataset. A single sample of such anomalous collection might not be considered anomalous on its own, but the occurrence together with the other samples in this collection is anomalous. An example can be see in Figure 3.3.

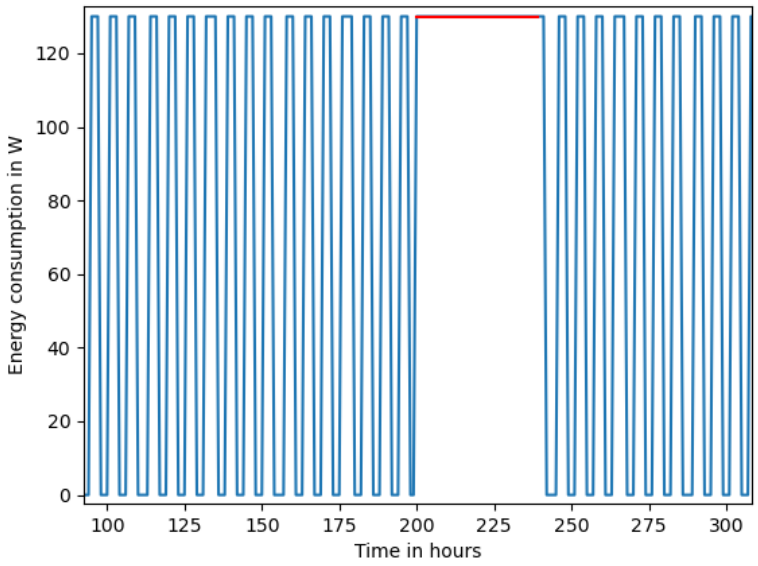

Figure 3.3: Synthetic energy consumption data containing a collective anomaly.

In this work we explore collective anomalies in the form of unexpected repeating sequences as well as contextual point anomalies.

3.3 Data Augmentation and Generation

When deploying a time series analysis tool in small to medium sized companies labelled anomalous data for automated detection method evaluation is usually not available. We still need this data to compare the performance indicators of different detection methods. One approach to obtain such data is to generate it based on a dataset that is considered normal [RESM17, SG19]. Normal historic data usually is available or can be collected from currently executed processes. Assuming that historic data exists we automatically generate anomalies in a given dataset in a similar fashion as Schneider et al. did in [SG19]. They defined so called *switches* which, when activated, override the initial value of the dataset with an anomalous value, for the time of its activation. We simulate such switches on our training datasets to inject controlled anomalies into them (see Algorithm 2 and Algorithm 3).

3.4 Anomaly Detection Methods

There are many established methods of anomaly detection in general, as well as specifically for time series [CBK09]. We use two more recent methods of the field, namely the *Matrix Profile* and *Long Short-Term Memory* neural network based anomaly detection and compare them to a simple heuristic based on Kernel Density Estimation.

3.4.1 Kernel Density Estimation

Kernel Density Estimation (KDE) is a nonparametric method to produce an estimate of the density function of a univariate probability distribution. Intuitively the task is to estimate the density of a finite dataset by generating a smoothed out histogram. A histogram of any set of real valued data is created by defining discrete bins or intervals and assigning each value to one of these bins. Instead of assigning each value to a bin, KDE assigns a density function (the kernel) to each value as seen in Figures 3.4 and 3.5.
These kernels are then added up to acquire an estimate of the density function of the whole dataset (see Figure 3.6) [PVG+11].

Finding Anomalies To find anomalies we have to choose some threshold for our estimated density function as illustrated in Figure 3.7. Values above the threshold are within the normal density, values below are not. The latter are considered anomalous.

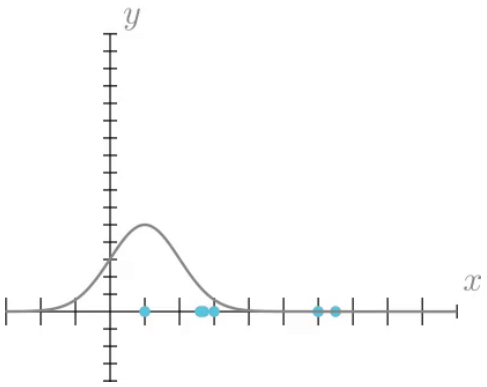

Figure 3.4: A kernel function with its maximum centred over a single point of a probability distribution.

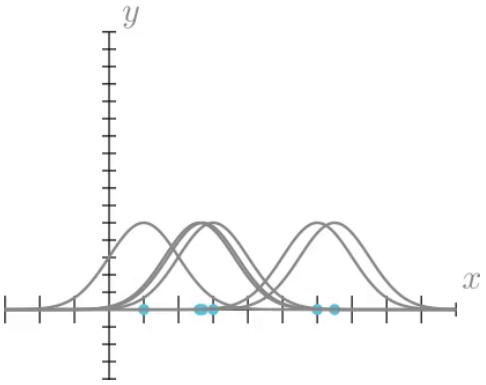

Figure 3.5: Kernel functions over all available points.

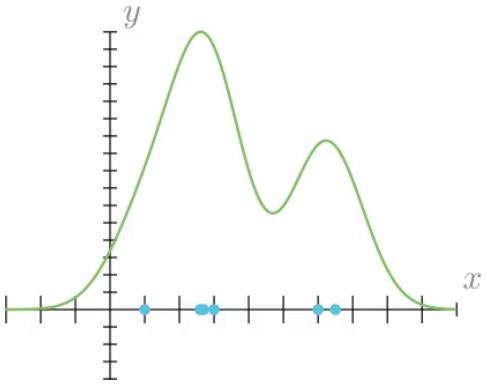

Figure 3.6: All the kernel functions added up provide a good estimate of the overall density function.

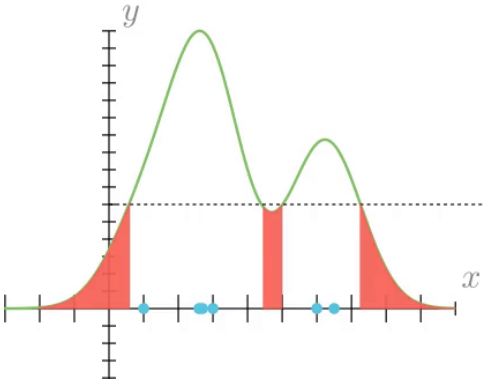

Figure 3.7: Adding a threshold, one can distinguish probable values from improbable ones.

3.4.2 Matrix Profiles

Matrix Profile is a relatively new similarity search algorithm based on the z-Normalized (see Appendix 1) euclidean distance [YZU+17, DBY+19]. It is used to efficiently provide a result for a similarity join problem, which can be generally described as: "Given a collection of data objects, retrieve the nearest neighbour for every object"[YZU+17]. In this case, the collection of data objects is a time series and every data object is a subsequence of values in this time series. As mentioned in Chapter 2, the Matrix Profile is a statistical method with high accuracy and little to no prior configuration. These attributes make Matrix Profile a reasonable choice for our prototype.
The Matrix Profile algorithm computes the z-Normalized euclidean distance of a signal with itself (or another signal), to find the nearest neighbours of all available subsequences. The following definitions were originally published in [YZU+17] and [MK21].

Definition 3.1. The **euclidean distance** of two different time series x and y:

$$d(x, y) = \sqrt{\sum_{i=1}^{n} (\hat{x}_i - \hat{y}_i)^2}$$

Where $x = \{x_1, x_2, \ldots, x_n\}$ and $\hat{x}_i$ is the z-Normalization of x_i. Similarly for y. □

Definition 3.2. Pearson's Correlation Coefficient is a measure of linear correlation of two different time series:

$$corr(x, y) = \frac{\sum_{i=1}^{n} x_i y_i - n\mu_x\mu_y}{n\sigma_x\sigma_y}$$

Where μ_x is the series' mean (see Appendix eq. 2), and σ_x is the series' standard deviation (see Appendix eq.3). It provides information on how strongly related these two time series are. □
The correlation coefficient relates to the euclidean distance as:

$$d(\hat{x}, \hat{y}) = \sqrt{2n(1 - corr(x, y))} \tag{3.1}$$

$$= \sqrt{2n(1 - \frac{\sum_{i=1}^{n} x_i y_i - n\mu_x\mu_y}{n\sigma_x\sigma_y})} \tag{3.2}$$

Using this relation Mueen et al. claim that calculating the Matrix Profile for the correlation coefficient and other time series measures is trivial, given an algorithm to compute the z-Normalized euclidean distance [YZU+17]. To verify this claim they provide the MASS (Mueen's Algorithm for Similarity Search) algorithm to compute the Distance Profile of a

time series efficiently.

Definition 3.3. The **Distance Profile** of a single time series can be computed by selecting a small[1] sequence, called *query* of length $m << n$ from the same or any other time series and computing the z-Normalized euclidean distances between the query and a sliding window of size m, which is dragged over the time series of length n. This results in the Distance Profile, which is a vector of distances $\vec{d} = d_1, ..., d_{n-m+1}$. □

$\mathbf{D}_1$	$\mathbf{D}_2$	...	$\mathbf{D}_{n-m+1}$
$d_{1,1}$	$d_{1,2}$	...	$d_{1,n-m+1}$
$d_{2,1}$	$d_{2,2}$	...	$d_{2,n-m+1}$
...	...	...	...
$d_{n-m+1,1}$	$d_{n-m+1,2}$	...	$d_{n-m+1,n-m+1}$

Table 3.1: A matrix containing several distance profiles as columns. Adapted from [MK21].

Definition 3.4. The **Matrix Profile** can be computed in two steps:

1. Create a Distance Profile for each possible window of size m to create a Distance Profile Matrix (see Table 3.1).

2. Select the minimum of each column $\mathbf{D}_i$ of the Matrix to create the Matrix Profile.

□

In Figure 3.8 we see two identical signals on the x and y axis surrounding a colourized distance matrix. One column in the matrix corresponds to one discrete point on the signal graph of the top x axis. For every single point on the x axis the euclidean distance to every point of the signal on the y axis is computed. The distance from each point of the x axis with each point on the y axis is computed via sliding a window (black box) of size 1 over the signal and plotted in the matrix. A blue square means low distance where red squares imply a high distance between the points in both sliding windows. The diagonal axis therefore contains only blue squares and mirrors the upper right triangle of the matrix to the lower left. Below the matrix is the *Matrix Profile* for the two identical signals. It contains the minimum value of each column's distance profile.

Finding Anomalies The computed Matrix Profile for some time series enables us to scan for patterns of many kinds. It provides the distance for each subsequence to its

[1]The relation << means *significantly smaller*.

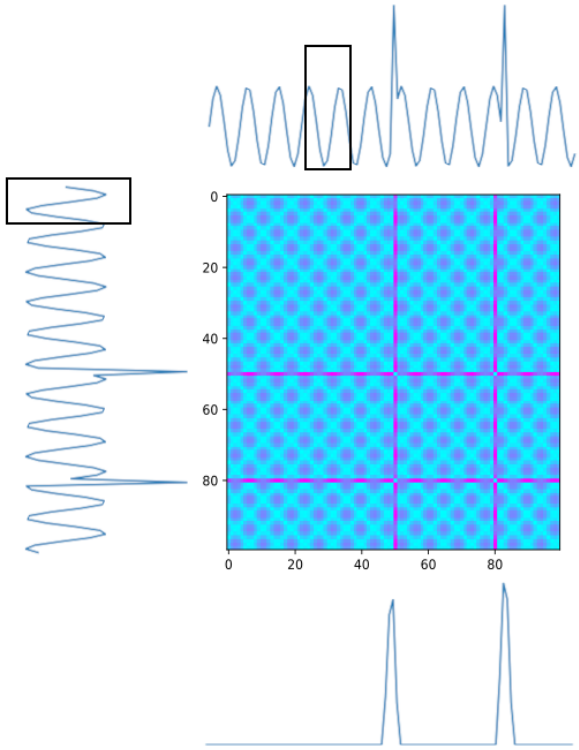

Figure 3.8: Illustrative example of a distance matrix of two identical signals and the corresponding Matrix Profile. Adapted from lecture slides [MK21].

nearest neighbour. Selecting a subsequence with maximum distance in this Matrix Profile is known as time series discord detection [YZU+17]. One can iterate from the maximum value to the next biggest value and so on to find every discord in a time series. How many such discords one finds depends on a threshold value similar to the one used in KDE.

3.4.3 Long Short-Term Memory

Long Short-Term Memory (LSTM) neural networks are recurrent neural networks (RNN) in which each neuron uses their own output as input for their next computation. They are especially useful for the prediction of sequential values (e.g. words in a sentence, stock price, etc.)[SS97, MVSA15].
A simple RNN consists of an input layer x a hidden layer h and an output layer y as seen in Figure 3.9.

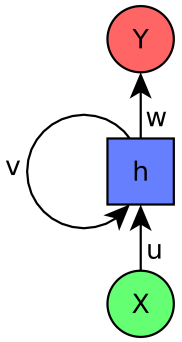

Figure 3.9: A simple RNN with a single hidden state. Adapted from [Ola21].

The input layer accepts an input vector (e.g. an encoded word) and passes it to the hidden layer. The hidden layer contains an activation function which is used to decide on what part of the input is passed to the next layer. Common activation functions are ReLU (Rectified Linear Unit), tanh and sigmoid functions (see Appendix 2). The connections between layers are weighted by u, v and w. Once an output (prediction) is produced, an error function computes the difference between the prediction and the expected output. With each new prediction the weights of the connections are changed to try and reduce this difference.
As can be seen in Figure 3.10, the weight v is the only weight connecting hidden states. This means the longer an input sequence is, the less influence the beginning of the input sequence has on v. This is known as the vanishing gradient problem. LSTM networks

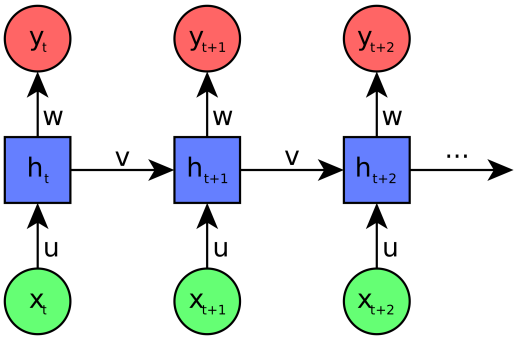

Figure 3.10: The same RNN after *unfolding* the feedback loop. Adapted from [Ola21].

mitigate the vanishing gradient problem by implementing a memory cell which keeps track of the long term development of input values and feeds it back into the network [MVSA15].

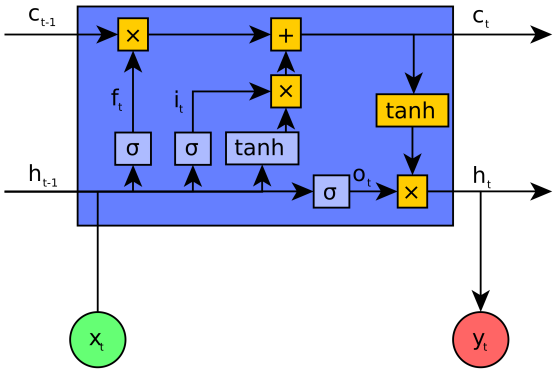

Figure 3.11: A single LSTM hidden state containing memory cell operations. Adapted from [Ola21].

Figure 3.11 shows such a memory cell. It consists of a forget gate (computing f_t), an input gate (i_t) and an output gate (o_t). These gates regulate the short and long term output of the cell by passing the input through sigmoid (σ) and hyperbolic tangent ($tanh$) activation functions. The cell state c_t and the hidden state h_t are passed to the next node of the unfolded network as seen in Figure 3.10.

Finding Anomalies Suppose we have a model that can predict new values for a given time series. Before a new value of a time series is generated we anticipate it with our

model prediction. The difference between the predicted value and the actual value is an indicator for anomalous behaviour [MVSA15]. How big this difference can be before a value is considered anomalous is yet again configured via a threshold similar to the one used in KDE and depends on the error (loss) computed during the training of the model.

3.5 Comparing multiple Time Series

From Section 3.1.3 we know which features can be used to characterize time series. The following section describes the means we use to compare the feature vectors of the collected, cleaned and reduced time series data (see Section 3.1) from multiple different data sources.

3.5.1 Clustering of Time Series

In unsupervised time series analysis, clustering techniques are used extensively.
A common application of clustering or bagging is the analysis of a single time series [HCW20]. We apply these techniques to distinguish multiple time series and the processes which emit them.
The general process of time series clustering can be described as follows. First, a number of features is extracted from a time series and stored in a feature vector. Then we place this vector in a high dimensional space and compute some form of distance from this vector to any other feature vector within the same space. Vectors with a small distance resemble similar underlying time series and should be grouped into the same cluster.

Number of Clusters A common problem in solving clustering tasks is that the number of clusters to find must be provided to the clustering algorithm beforehand. Given the variety of data sources and the desired autonomy of the system, no assumptions concerning the number of clusters can and should be provided by the user.
There are many different methods which estimate the number of available clusters in a given dataset. Such as the *Elbow Method* and *x-means-clustering* [Dan02].
These techniques require the user to either provide an estimate for the expected number of clusters on their own or to visually inspect the estimators' results. This user interaction is not required when using DBSCAN (Density Based Spatial Clustering of Applications with Noise) or BIRCH (Balanced Iterative Reducing and Clustering using Hierarchies), which rely on parameters that can be configured initially (and potentially dynamically) with respect to the computed feature vectors. These methods do not rely on labelled data or any form of prior training, which makes them suitable for our task. The following

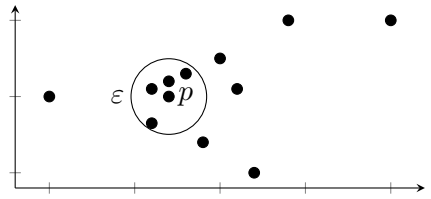

Figure 3.12: Eps-neighbourhood for some arbitrary data point p.

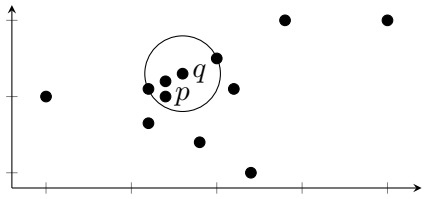

Figure 3.13: Point p is directly density-reachable from point q.

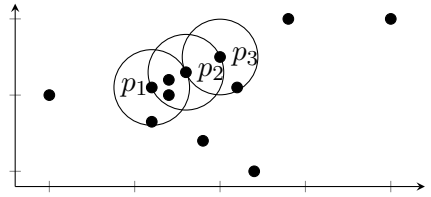

Figure 3.14: Point p_3 is density-reachable via p_2 from point p_1.

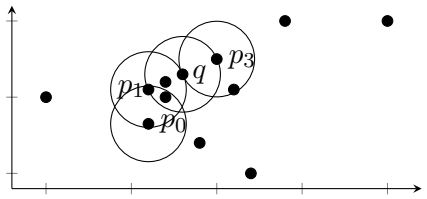

Figure 3.15: Point p_0 is density-connected to p_3 via q.

sections contain information on these clustering techniques and we further evaluate their usefulness regarding our implementation in Chapter 5.

3.5.2 DBSCAN

In clustering we want to use spatial relations of objects (points) to form groups of similar objects. The DBSCAN algorithm accomplishes this task using the density which is an inherent attribute of any spatial representation $\mathcal{D}$ of multiple data objects. The points surrounding another point in some fixed radius determine this point's neighbourhood, which is a subset of $\mathcal{D}$ and inherits the density property. The following algorithm and its preliminary definitions were originally published in [EKSX96] and are summarized here to provide theoretical background knowledge for explanations later in this thesis.

Definition 3.5. The **Eps-neighbourhood** $N_\varepsilon(p)$ of a point p is defined as:

$$N_\varepsilon(p) = \{q \in \mathcal{D} \mid \operatorname{dist}(p, q) \leq \varepsilon\}$$

Where ε refers to the neighbourhood radius and $\operatorname{dist}(p, q)$ is some function to compute the distance between p and q. □

The core idea is that for a point to belong to a cluster the number of points surrounding it (Eps-neighbourhood) has to pass a certain threshold T_{MinPts}. But there are points at the border of a cluster which might not satisfy this threshold condition. This necessitates a

second condition to include these points in a cluster. The original description of DBSCAN contains a distinction between core points and border points. Core points have to have a neighbourhood size that passes the threshold T_{MinPts} to be considered as such.

Definition 3.6. Point p is considered a **core point** if:

$$|N_\varepsilon(p)| \geq T_{MinPts}$$

holds. □

If, for example $T_{MinPts} = 4$, point p in Figure 3.12 would be considered a core point. Border points are no core point themselves but are directly density-reachable from any core point. This relation is not symmetric. Meaning that any point belonging to the neighbourhood of a core point is directly density-reachable from this core point but not necessarily the other way around.

Definition 3.7. A point p is **directly density-reachable** from a point q if the following conditions are met:

1. $p \in N_\varepsilon(q)$
2. $|N_\varepsilon(q)| \geq T_{MinPts}$

□

Point p in Figure 3.13 is directly density-reachable form point q, when $T_{MinPts} = 4$, since p is in $N_\varepsilon(q)$ and $|N_\varepsilon(q)| \geq 4$.

Definition 3.8. For point p to be considered a **border point** it would have to satisfy a third condition

1. p is directly density-rechable from a core point q
2. $|N_\varepsilon(p)| < T_{MinPts}$

□

Else p would just be another core point which happens to be directly density-reachable from q. Point p_3 in Figure 3.14 is a border point that is density-reachable from p_1 and directly density-reachable from p_2.

Definition 3.9. The transitive expansion of the directly density-reachable relation is the **density-reachable** relation. Thus a point p_n is considered density-reachable from point p_1 if there is a chain of points $p_1, \ldots, p_n$ in which each p_{i+1} is directly density-reachable from p_i for all $1 \leq i \leq n$. □

If p_1 and p_n are border points in the same cluster C there cannot be such path of directly density-reachable points connecting them because the core point condition (see Definition 3.6) cannot hold for p_1. To connect two border points there must be a core point o in C that is density-reachable for p_1 as well as p_n.

Definition 3.10. The points p_1 and p_n are **density-connected**, if and only if there exists a core point q such that p_1 and p_n are both density-reachable from q. □
In Figure 3.15 point p_3 is not density-reachable from p_0 for $T_{MinPts} = 4$ since $|N_\varepsilon(p_0)| \geq T_{MinPts}$ does not hold, but these two points are density-connected via q and p_1.

Definition 3.11. A **cluster** C can now be defined as a non-empty subset of dataset $\mathcal{D}$ where the following conditions are met:

1. $\forall p, q:$ if $p \in C$ and q is density-reachable from p, then $q \in C$
2. $\forall p, q \in C:$ p is density-connected to q

□

Definition 3.12. Any point not belonging to a cluster is considered **noise**:

$$\text{noise} \;=\; \{p \in \mathcal{D} \mid \forall i : p \notin C_i\}$$

□

So given the parameters ε and T_{MinPts}, DBSCAN can detect a cluster C in two steps. First select any point p_{core} from $\mathcal{D}$ that satisfies the core-point condition (see Definition 3.6). Second add all points to C which are density-connected to p_{core}. The Algorithm 1 shows how this principle can be used to assign a *clusterId* to each point in a given dataset.

Algorithm 1: DBSCAN clustering algorithm adapted from [EKSX96].

```
1  Input: Set of points 𝒟 = {p_0, p_1, ..., p_n} with n ∈ ℕ
2  Input: Neighbourhood radius ε
3  Input: Core point condition threshold T_MinPts.
4  cId ← nextId(noise)
5  for p_i ∈ 𝒟 do
6      if p_i.clusterId == unclassified then
7          neighbourhood ← N_ε(p_i)
8          if |neighbourhood| < T_MinPts then
9              p_i.clusterId ← noise
10         else
11             for q_j ∈ neighbourhood do
12                 q_j.clusterId ← cId
13             neighbourhood.delete(p_i)
14             while |neighbourhood| > 0 do
15                 qNeighbourhood ← N_ε(q_k)
16                 if |qNeighbourhood| ≥ T_MinPts then
17                     for o_h ∈ qNeighbourhood do
18                         if o_h.clusterId ∈ {unclassified,noise} then
19                             if o_h.clusterId ==unclassified then
20                                 neighbourhood ← neighbourhood ∪ {o_h}
21                             o_h.clusterId ← cId
22                 neighbourhood.delete(q_k)
23             cId ← nextId(cId)
24 return 𝒟
```

Regarding our use case (the processing of streaming data) the DBSCAN algorithm has a significant drawback: It does not work iteratively. If any point is added to the dataset after the clustering process the whole clustering must be computed anew. An evaluation and comparison of DBSCAN and another clustering method (BIRCH) is given in Chapter 5.

3.5.3 BIRCH

BIRCH describes a tree data structure to iteratively store and cluster large amounts of data points. The following section provides a brief summary of the work presented by Zhang et al. in [ZRL96]. Iterative clustering is useful for us because of two system specifications. First we want to handle streaming data where new subsets of the data arrive iteratively. Secondly we might want to add additional data sources to the system without having to restart the whole clustering process. If we expect such a new data source to be similar to other sources already in use we can add this new source and would expect it to be grouped and analysed similar to already known data sources. A completely new data source dissimilar to all known data sources could just be iteratively inserted into the BIRCH tree data structure. The following algorithm and definitions originate from [ZRL96].

Definition 3.13. A single data **point** p_n consists of a single vector containing real values.

$$p_n = \langle v_1, v_2, \ldots, v_k \rangle$$

Where $k \in \mathbb{N}$ and $v_1, \ldots, v_k \in \mathbb{R}$. □

Initially BIRCH scans the whole dataset and builds a tree data structure representing the dataset by iteratively inserting data points. This tree then contains leaf nodes which represent subsets with a high density of data points that are assumed to be clusters.

Definition 3.14. A **cluster** C_i consists of several data points $p_1, p_2 \ldots, p_N$.

$$C_i = \{p_1, p_2, \ldots, p_N \mid N \in \mathbb{N}\}$$

Where $i \in \mathbb{N}$. □

Then it takes a representative (centroid) of each small cluster to run a clustering on this subset of representatives to build larger clusters consisting of multiple subclusters.

Definition 3.15. The **centroid** p_0 of a cluster is the mean value of all its points.

$$p_0 = \frac{\sum_{i=1}^{N} p_i}{N}$$

□

The algorithm used for the for the clustering of representatives is no part of BIRCH itself and there are different variants of BIRCH working with different clustering algorithms [ZRL96].
At the core of this clustering data structure are the cluster features that store information on clusters. Every node of the tree contains at least one cluster feature and a link to the corresponding child node. For leaf nodes the child nodes are subclusters. For non-leaf nodes the child nodes contain one or more cluster features.

Definition 3.16. A **cluster feature** (cf) $\mathrm{cf}(C)$ is a triple of the number of data points, the linear sum of these data points and their squared sum.

$$\mathrm{cf}(C_i) = (N, LS_i, SS_i)$$

Where $LS_i = \sum_{j=1}^{N} p_j$ and $SS_i = \sum_{j=1}^{N} {p_j}^2$ □
Such cluster feature is sufficient to characterize a cluster and distinguish it from other clusters.

Definition 3.17. A **cf-tree** is a height-balanced tree data structure to manage cluster features. Two parameters determine the trees' height and node distribution namely the branching factor $B \in \mathbb{N}$ and the threshold $T \in \mathbb{R}$. B determines how many children a single node might have. T is a value for a threshold condition, that must be fulfilled by leaf nodes. By default T represents the maximum diameter, that a subcluster in a leaf node might have. It can be replaced by other metrics such as the radius of the subcluster.□

Insertion of a new entry into the tree happens in three to four steps:

1. Find the appropriate leaf node:
   By comparing the new entry to the centroids of the available leaf nodes the cluster closest to the new entry can be found.

2. Insert the entry into the leaf node:
   Once the entry reaches a leaf node the new entry will be 'absorbed' into the leaf node. If the 'absorption' would cause the leaf node to violate the threshold condition T the leaf node must be split.

3. Update the path that leads to this leaf node:
   Once the new entry has found a place in the tree the path leading from the root to this place must be updated. Each node on this path needs to add the new number of points linear and square sum to their respective cluster features.

4. If a leaf node is split due to spatial limitations (threshold T) search for other leaf nodes to merge to keep the number of leaf nodes small and delay reaching the threshold.

Definition 3.18. The **radius** R and **diameter** D of a cluster are defined as

$$R = \left(\frac{\sum_{i=1}^{N} (p_i - p_0)^2}{N} \right)^{\frac{1}{2}}$$
$$D = \left(\frac{\sum_{i=1}^{N} \sum_{j=1}^{N} (p_i - p_j)^2}{N(N-1)} \right)^{\frac{1}{2}}$$

□

The main algorithm consists of four phases, of which two (Phase 2 and 4) are optional but might increase performance.

1. Scan the available dataset once and build a cf-tree.
2. Scan the leaves of the built tree for possible merges and reduce the size of the tree if possible.
3. Use a clustering algorithm directly on the cluster features representing subclusters.
4. Centroids of the leaves of the cf-tree are used to build final clusters.

Some leaf nodes might show a low density of entries compared to other leaf nodes even after the clustering and merging operations. BIRCH considers these leaf nodes outliers. In our case these outliers would be indicators of datasets which are not similar to any existing dataset.

3.6 Meta-Learning

We have described algorithms to find anomalies in time series data and elaborated how characteristics of time series can be used to group and distinguish them. We can now combine these methods and implement a system for Meta-Learning. Meta-Learning generally describes systems that are learning how to learn. Its ultimate goal is to design a system which is able to choose an optimal classifier for any given dataset. Such systems, for example RapidMiner (prev. YALE) [MWK+06], are used widely[2] to speed up data analysis workflows or to enable data analysis where no data analyst or machine learning

[2] According to polls of experts in the field https://www.kdnuggets.com/2019/05/poll-top-data-science-machine-learning-platforms.html

expert can be consulted. First articles considering Meta-Learning as a tool for improved anomaly detection were published recently [ZRA20].

3.6.1 Method Selection

To select a fitting anomaly detection method for any given dataset we must be able to evaluate the performance of a classifier and compare this performance to other detection methods. In state-of-the-art Systems this selection is based on predicting the classification accuracies of different classifiers [RSG+14]. We use this approach to select methods of anomaly detection. Although we do not predict classification accuracies based on previous real dataset classifications as in [ZRA20], but estimate the performance of different anomaly detection methods for some time series data by generating anomalous data and executing the detection method on this data.

3.6.2 Method Evaluation

There are several well established performance indicators to compare different anomaly detection methods used in many different applications and publications. The four performance indicators F1-Score, precision, recall and accuracy are commonly used in anomaly detection evaluation or classification evaluation in general [DAFS19, LASE18, HCH19].

Definition 3.19. Precision denotes the percentage of true positive t_p classifications in relation to all the positive classifications $t_p + f_p$ including false positives f_p and thereby describes how precise the notification for positive classifications is.

$$precision = \frac{t_p}{t_p + f_p}$$

□

Definition 3.20. Recall relates the number of samples correctly marked as positive t_p with the samples that should have been marked as positive but were not f_n.

$$recall = \frac{t_p}{t_p + f_n}$$

□

Definition 3.21. The **accuracy** takes into account all the correct classifications $t_p + t_n$ (true positives and true negatives) and sets them in relation with the whole set of classified

data.

$$accuracy = \frac{t_p + t_n}{t_p + f_p + t_n + f_n}$$

□

Definition 3.22. The **F1-Score** combines precision and recall into one single indicator.

$$F_1 = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

□

Each evaluation of an anomaly detection method produces some value for each of these performance indicators. The accuracy indicator on its own is no good metric since we handle very unbalanced data. In a large dataset there might be only a single anomaly and a detection method labelling every sample as benign would achieve high accuracy because there is only one false negative. The F1-Score mitigates this effect of very unbalanced data and makes values close to 1 harder to reach. The F1-Score is used to compare detection methods in the learning phase of the prototype system. We also use these indicators in our overall evaluation of the system in Chapter 5.

4 Implementation

In the following section we describe the general system architecture and provide explanations of the single modules, their functionality and interactions. We then focus on the most relevant modules regarding our research questions from Chapter 1. Therefore we first explain how we generate anomalies for each individual data source. Then we elaborate on our implementation and use of offline and online clustering. Finally we provide details on the detection of anomalies and the ways data emitted by data sources can be handed to the system. Since anomaly detection is a huge field with many methods and application domains [CBK09] we have to limit the expected capabilities of our prototype implementation. Some of the most impactful limitations are:

- We limit our prototype to one dimensional, sequential data only.
- We use a small set of detection methods, each with an unique approach to the problem.
- Unlike other studies (see Chapter 2) we use a fixed set of pre-processing steps, rather than implementing a dynamic selection of the best possible pre-processing setup.

4.1 Architecture and Components

Figure 4.1 illustrates the general architecture of our prototype system with its main components and their respective dependencies. The figure consists of three main areas: The *Data Processing* area (light blue), the *Training* area (dark blue) and the *Test/Usage* area (green). The *Data Processing* area handles data collection and preparation as explained in Section 3.1. Assume we want to monitor a temperature sensor data stream. To bootstrap the learning process, an ICS might provide historical log data or we can create our own log data by collecting current samples from active data sources. Either of these actions is performed via a_1. The preprocessing (see Section 3.1.2) is executed in the same step. We then compute some characteristics of the collected data (b_1) and proceed by grouping it with similar data in a cluster. For half of the collected data (b_2) we generate some random anomalies and inject them into the given data. The other half is considered benign data and is later used for training the detection methods. After this, we select one representative data source from each cluster (c_1) and attach all of the available methods to these representatives (d). In the Method Evaluation module we use the data containing

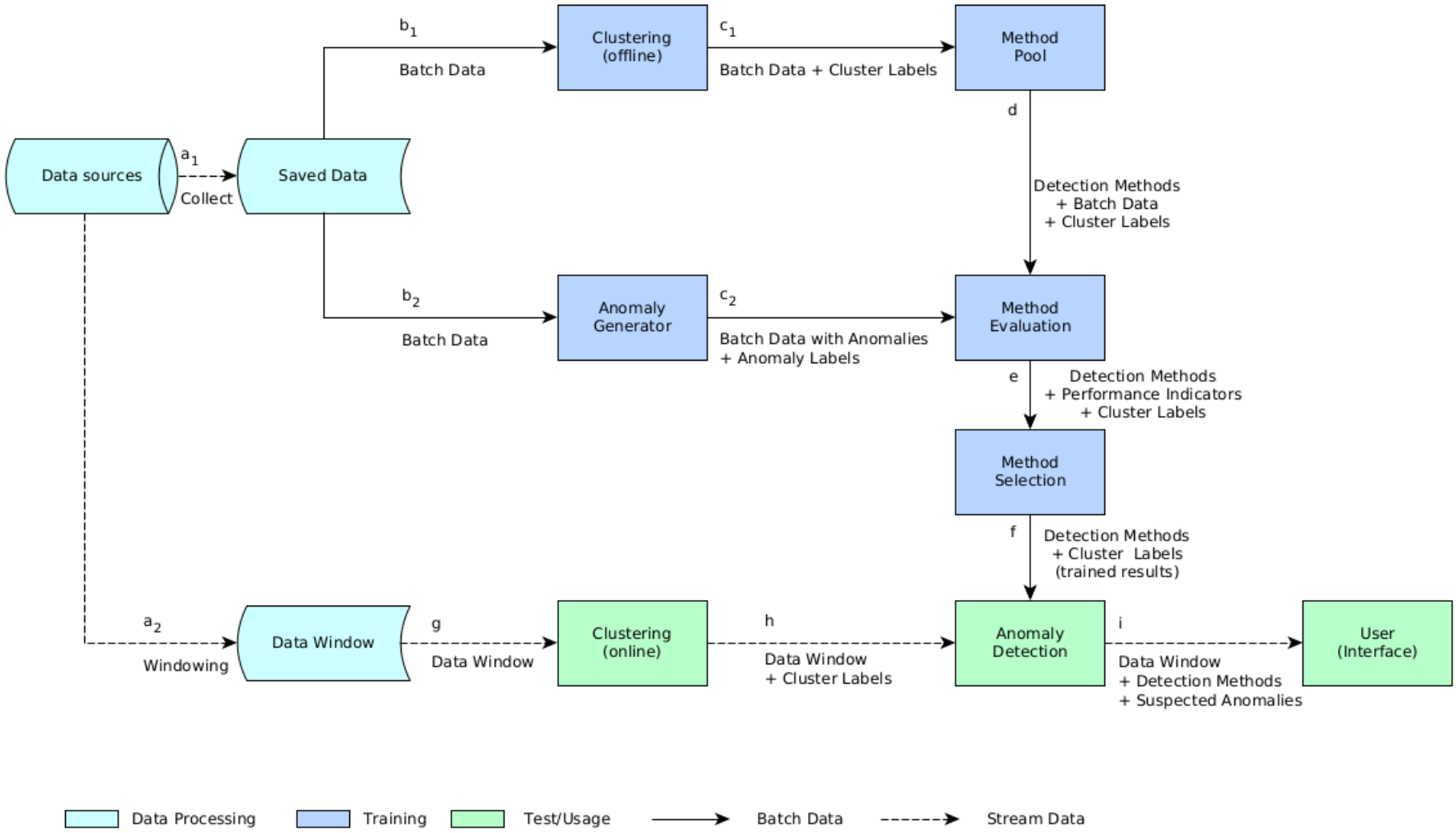

Figure 4.1: General system architecture.

the generated anomalies (c_2) to evaluate the available methods. Then (e) we select the best method for anomaly detection on this kind of data and (if necessary) train this method to learn normal behaviour using the benign data.

After training the system this way, we use the trained results (f) for live monitoring of the temperature sensor. We collect subsequent windows of live data from the sensor (a_2) and compute the characteristics (g) of each window (online) to group it with similar data in a cluster. Ideally our temperature sensor data would get the same label as it had before when clustering historical log data. If that is not the case, we inform the user and assign (and if necessary train) a new detection method on the new data. With the cluster information (h) and the trained results (f) we can select the best anomaly detection algorithm quickly and run it on the data window. The results (i) are then displayed in a message to the user (interface).
The following paragraphs provide a brief description of the implemented modules and their interplay with other modules.

Module: Clustering (offline) This module groups the previously logged (hence 'offline') time series by their characteristics (see Appendix 3). Each time series provided by the available data sources is analysed and its characteristic features are extracted and placed into a feature space. In this feature space, the distance between different feature vectors is used as a measure of similarity. A small distance between two vectors indicates that their emitted data is similar and they are grouped together.
Using this method, clusters of similar data sources form. We evaluate the sklearn ([PVG+11]) implementations of DBSCAN (see Section 3.5.2) and BIRCH (see Section 3.5.3) clustering algorithms for our prototype system (see Section 5.4.1). A detailed description of their usage is given in Section 4.2.3.

Module: Anomaly Generator The Anomaly Generator transforms an unlabelled time series containing no anomalies into a time series containing labelled anomalies. This transformation is needed to evaluate any of the anomaly detection methods available in the Method Pool (see **Module: Method Evaluation**). Since for each data source some different single value might be considered an anomaly we cannot simply inject some static values into the data and expect them to be anomalous regarding this data source. We use some simple algorithms similar to the *switches* used in [SG19] to create anomalies based on the given time series. The details of these algorithms and their implications are given in Section 4.2.1.

Module: Method Pool The Method Pool contains selectable methods for anomaly detection. Each method provides an interface that accepts configuration parameters and time series data. We use python implementations of *Long Short-Term Memory* [AAB+15], *Matrix Profile* [BOBM20] and *Kernel Density Estimation* [PVG+11] to detect anomalies.

Module: Method Evaluation The Method Evaluation module receives labelled time series data from the Anomaly Generator module and uses the data to evaluate the proficiency of the available methods. Since we expect there to be many data sources we do not want to evaluate each method for each data source. Here the output of the Clustering module comes into play. We evaluate one representative dataset from each cluster of similar datasets. This reduces the computational cost of method evaluation significantly, while still selecting the best detection method under the right circumstances (see Section 5.5.2). Since we generated our own anomalies and injected them into the time series, we know how many anomalies there should be (true positives) and if a found anomaly is normal data (false positives). As our key performance indicator we use the F1-Score (see Section 3.6.2) that is widely used in literature on evaluation of classifiers and anomaly detection methods. The result of this module is a mapping of F1-Scores to detection methods for each representative dataset.

Module: Method Selection The result of the method evaluation is passed to the Method Selection module which selects the detection method with the highest performance indicator value. This method is trained (if needed) on the previously collected normal (without injected anomalies) data.

Module: Clustering (online) When a new time series window arrives we need to determine which cluster it belongs to. We compute the same features as in the Clustering (offline) module and compare this feature vector to the ones already available from offline clustering. If the new time series window contains data that is similar to data previously emitted by the observed data source, the previously used anomaly detection method will remain and try to detect anomalies in this new time series. If the online clustering results in a new cluster label for the observed data source, the detection method will change according to this new label. A detailed description of this process is given in Section 4.2.3.

Module: Anomaly Detection The Anomaly detection module uses the gathered information on data source clusters and their respective anomaly detection methods to actually analyse process data regarding anomalies. Newly arriving time series windows and their

cluster labels are passed on to this module which then executes the detection method assigned to the given cluster label for each available data source.

4.2 Implementation Details

In this section we elaborate on some of the modules and their processing details. We first take a closer look at the Anomaly Generation module and then explain the clustering process and the differences between the online and offline clustering modules. We also provide details on the Method Evaluation module.

4.2.1 Anomaly Generation

In a similar approach to the switches mentioned in Section 3.3, we will modify a given dataset and save the modified time series data as well as the modification parameters in addition to the original time series data.
We create controlled anomalies by adding extreme values randomly to an otherwise normal dataset. In our case extreme values are created by doubling the maximum value of the normal dataset.

Algorithm 2: Maximum doubling at random indices.

Input: Data sequence $\mathcal{D} = \langle d_0, d_1, \ldots, d_{n-1} \rangle \in \mathbb{R}^\star$ with $n \in \mathbb{N}$ and $d_i \in \mathbb{R}$
Input: Number of anomalies $m \in \mathbb{N}$ with $m \leq n$
randomly select m indices $\mathcal{I} = \langle i_0, i_1, \ldots, i_{m-1} \rangle$ with
  $i \in \{0, 1, \ldots, n-1\}$ and $i_j \neq i_k \; \forall \; j, k \in \{0, 1, \ldots, m-1\}$ where $j \neq k$
$x \leftarrow max(\mathcal{D}) \cdot 2$
for *every index* i *in* $\mathcal{I}$ **do**
  $d_i \leftarrow x$
return $\mathcal{D}, \mathcal{I}$

Algorithm 2 describes this process. At first, we select m unique indices randomly from the available indices of data sequence $\mathcal{D}$. Each data value $d \in \mathcal{D}$ at one of those selected indices is replaced by a new value $max(\mathcal{D}) \cdot 2$. The algorithm returns the modified data sequence $\mathcal{D}$ and the list of randomly selected indices $\mathcal{I}$.
Another implemented approach to create controlled anomalies consists of the random doubling of subsequences. We choose a random, fixed length subsequence of a given time series, copy the subsequence data points and override another random subsequence of the same length with these copied data points.
Figure 4.2 illustrates Algorithm 3 with initial parameters $\mathcal{D} = \langle d_0, d_1, \ldots, d_{2000} \rangle$ with d_t being some sine function at time step t and $l = 0.1$, leaving us with $\lambda = \lfloor 2000 \cdot 0.1 \rfloor = 200$.

Algorithm 3: doubleSequence($\mathcal{D}$,l) Sequence doubling at random indices.

Input: Data sequence $\mathcal{D} = \langle d_0, d_1, \ldots, d_n \rangle \in \mathbb{R}^\star$ with $n \in \mathbb{N}$ and $d_i \in \mathbb{R}$
Input: Relative length of subsequence $0 < l < 0.5$
$\lambda \leftarrow \lfloor n \cdot l \rfloor$
randomly select sequence $S = \langle s, s+1, \ldots, s+\lambda \rangle$ from $\mathcal{D}$
randomly select start index u of sequence to replace, where $u < s - \lambda$ or $s + \lambda < u$
for *every index* $j \in \{s, s+1, \ldots, s+\lambda\}$ *and* $k \in \{u, u+1, \ldots, u+\lambda\}$ **do**
    $d_k \leftarrow d_j$
return $\mathcal{D}, u, \lambda$

The randomly selected sequence is coloured green in the figure and starts at the time step $s = 400$. This generates $S = \langle 400, 401, \ldots, 600 \rangle$. The randomly selected start index $u = 1000$ satisfies the second condition $s + \lambda < u$ because $600 < 1000$. The for loop now replaces each value of $\mathcal{D}$ between the indices 1000 and 1200 with the values between the indices 400 and 600. This replacement is highlighted in red colour.
In this case of a very repetitive time series (sine), a doubled sequence results in two point anomalies at the beginning and the end of the inserted sequence. Doubling a sequence of a more complex time series might shift the anomaly type towards collective anomalies or even contextual collective anomalies as explained in Section 3.2.1.

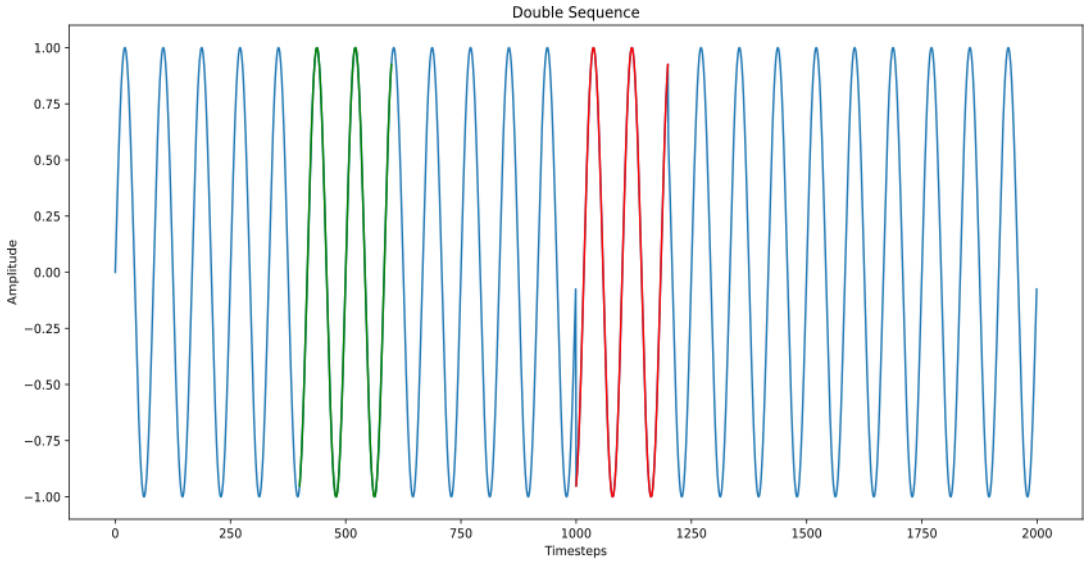

Figure 4.2: An illustration of the double sequence algorithm.

4.2.2 Method Evaluation

In the Method Evaluation module we use the previously generated anomalies to evaluate the available anomaly detection methods. We do not evaluate each detection method for every single data source but randomly select a representative data source for each detected cluster (see Section 4.2.3) to evaluate the available methods on. The core idea is here, in the ICS context there might be hundreds or thousands of different data sources. Running this evaluation on each of the data sources might consume a large amount of time. By clustering similar data sources, evaluating representatives of these clusters and using the evaluation results for each data source of the cluster can be much faster and might produce the same outcome (see Chapter 5).
Anomaly detection methods are evaluated differently depending on how they work. Methods that have to be trained on normal data (e.g. KDE and LSTM) will be evaluated via simple cross-validation. We split the available normal data into two subsets. One is used to train the detection method model, the other contains injected anomalies and is used for validation. For some methods of the method pool (e.g. Matrix Profile) the training is omitted and so is the cross-validation. We just feed the anomaly injected data to the method and compute the performance indicators for the results.
Figure 4.3 shows a flowchart for the evaluation of a single data source and a single method. This process is executed for each representative and each detection method respectively. A loop encloses the whole evaluation process since we want to mitigate possible one-off evaluations by building the mean over this loop's results.
Each run of the loop, we first generate new anomalies randomly for the data source d selected for evaluation. We then check if the selected method m needs some form of training. If it does, we train it on the available training data of d before trying to detect the generated anomalies with the selected method. By building the intersection of the true (generated) anomalies and the found anomalies we can compute the true and false positives as well as true and false negatives of method m for data source d. With these values we are finally able to compute the precision, recall, F1-Score and accuracy of the method m for the data source d. These values are summed up and divided by the iterations of the loop to build a mean. These mean values are saved as results of the method evaluation.

Windowed Index Intersection All implemented anomaly detection methods return time series indices where potential anomalies were found as well as the number of surrounding indices, which might be affected by this anomaly. This is because some methods, especially Matrix Profile, do not compute exact matches but rather return regions of time series where anomalies are highly probable. When evaluating the detection methods, we compare the list of generated (known) anomaly indices with the list of found anomaly

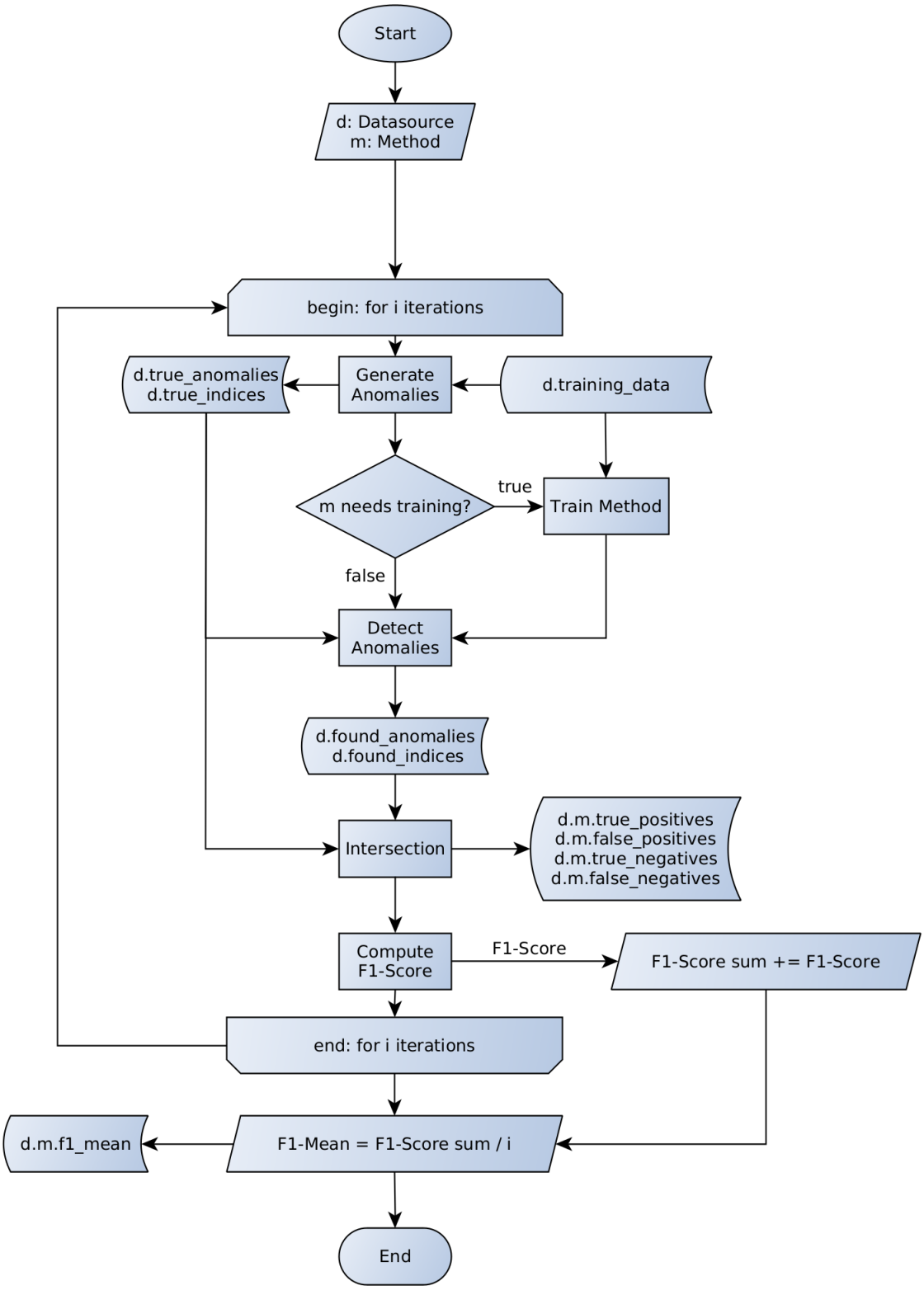

Figure 4.3: A flowchart showing the method evaluation process.

indices. To allow for detection methods to define a region or window surrounding some found anomaly we implemented a helper function. This function builds an intersection of two lists within a window of uncertainty.
Suppose a detection method found anomalies at the indices $found = \langle 12, 40, 500 \rangle$ and gives the sizes of windows surrounding those anomalies as $windows = \langle 1, 0, 3 \rangle$. The generated anomalies in this example are $generated = \langle 11, 40, 550 \rangle$. The windowed index intersection first extends the list of generated anomaly indices according to the given windows to $generated = \langle 10, \mathbf{11}, 12, \mathbf{40}, 547, 548, 549, \mathbf{550}, 551, 552, 553 \rangle$. Then we compute the intersection of *found* and our extended *generated*:

$$found \cap generated = \langle 12, 40 \rangle$$

This indicates that the found indices 12 and 40 are considered true positives in the evaluation module, while 500 was too far away from the true index 550 even with a window of ±3, so it will be considered a false positive and 550 a false negative. A *windows* list containing only zeros will cause the function to only return exact matches. The implemented KDE and LSTM based anomaly detection methods return such zero lists. Although this helper function allows for detection methods to have non-exact matches counted as positive, the window of uncertainty leads to many false positives as can be seen in Chapter 5.

4.2.3 Clustering

Since we want to measure the similarity of many (possibly thousands) different data sources, we need efficient and scalable clustering algorithms. We work with two different clustering processes: offline and online clustering. In the following paragraphs we explain the details, differences and similarities of these two processes.

Offline Clustering is executed during the learning phase of the system (see Figure 4.1). The goal is to group similar data sources based on the characteristic features of the time series data these data sources emit. In offline clustering we use previously recorded data (training data) of these data sources to reach this goal. As can be seen in Figure 4.4, the first step is to generate the characteristic features from existing training data with the *get_features* action. These features are saved for later use and simultaneously passed to BIRCH (see Section 3.5.3). BIRCH builds a *cf-tree*, which is saved as *BIRCH Model* and creates our cluster labels. The cluster labels are attached to each data source object as an attribute so that we can refer to them later. Once the offline clustering is done the system moves on to method selection and evaluation as explained in Section 4.1.

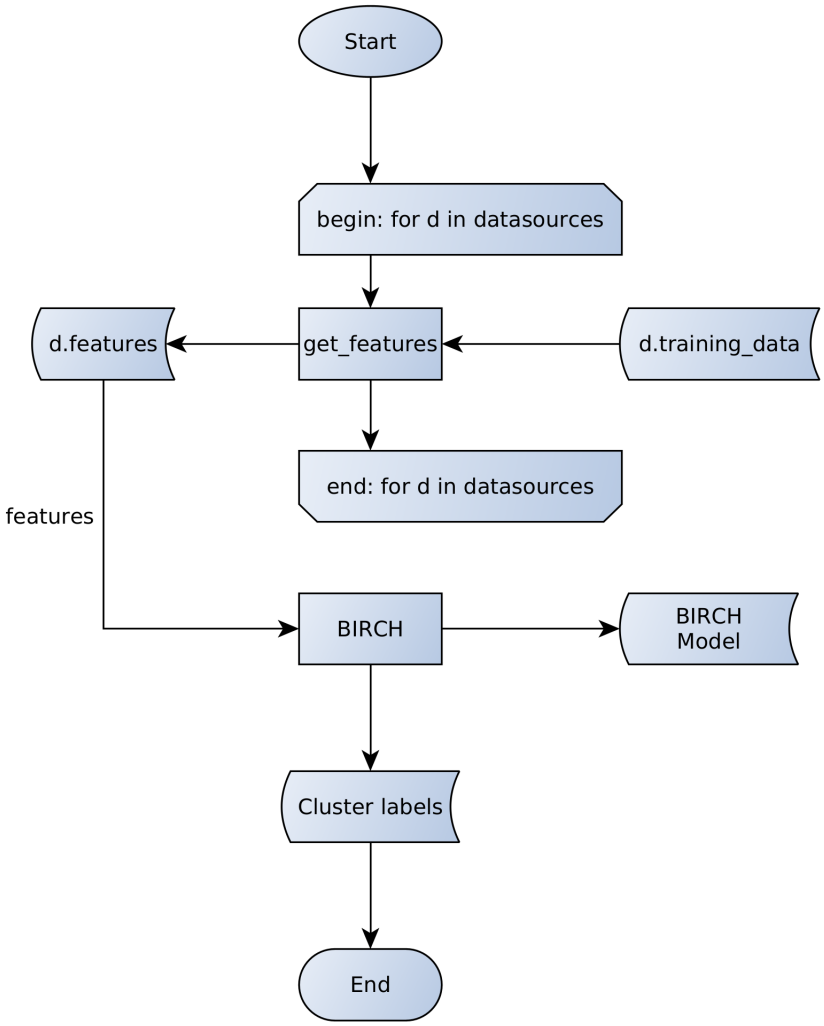

Figure 4.4: A flowchart showing the offline clustering process.

Online Clustering is executed in the main program loop. Since the data sources continuously emit new values we need to continuously analyse the data regarding their assigned cluster label and their respective anomaly detection method. To do so, in each execution of the main program loop we iterate through all available data sources and gather newly emitted data from a window of fixed size in an *accumulate* method (see Figure 4.5). Next we extract features from this gathered data much the same as in offline clustering. Then we employ the previously trained and saved *BIRCH Model* to compute the cluster label of the data source cluster to which the current data source belongs.
At this point we have a new cluster label for the newly emitted data from the current data source and an old cluster label from previous analysis (previous loop or initial training). In the following step we compare these two labels. If they are equal this means that the newly emitted data is similar to data previously emitted by this data source and similar data sources. We assume that the anomaly detection method for this cluster of data sources is still applicable to the current data source. If the two labels differ we have to assume that the currently emitted data belongs to a different cluster of data sources and their anomaly detection method must be applied. Therefore we change the currently used anomaly detection method to the one of the newly attributed cluster and send a message to the user, stating that a data source changed its cluster. This allows us to track behavioural changes over long time periods. A change of cluster labels might be considered an anomaly on its own.
Since it is also possible that this newly emitted data does not belong to any known cluster, we can use the iterative insertion capabilities of BIRCH to create new cluster labels without stopping the main program loop and restarting the clustering.

4.3 Detection

We integrated three methods to detect anomalies: Kernel Density Estimation, Matrix Profile and Long Short-Term Memory. Their theoretic background is covered in Section 3.4. The following paragraphs provide details on the libraries used to implement these methods and how they are configured. In general our implementation provides one or two interfaces to these methods. A detection interface to detect anomalies and a training interface if the method requires some form of training.

Kernel Density Estimation For KDE we use *scikit-learn* and its *KernelDensity* class [PVG+11]. Since in KDE the density has to be estimated on the available normal time series data, we implement a training function to train a model on such data. It first scales the data to zero mean and unit variance (see z-Normalization in Appendix Formula 1)

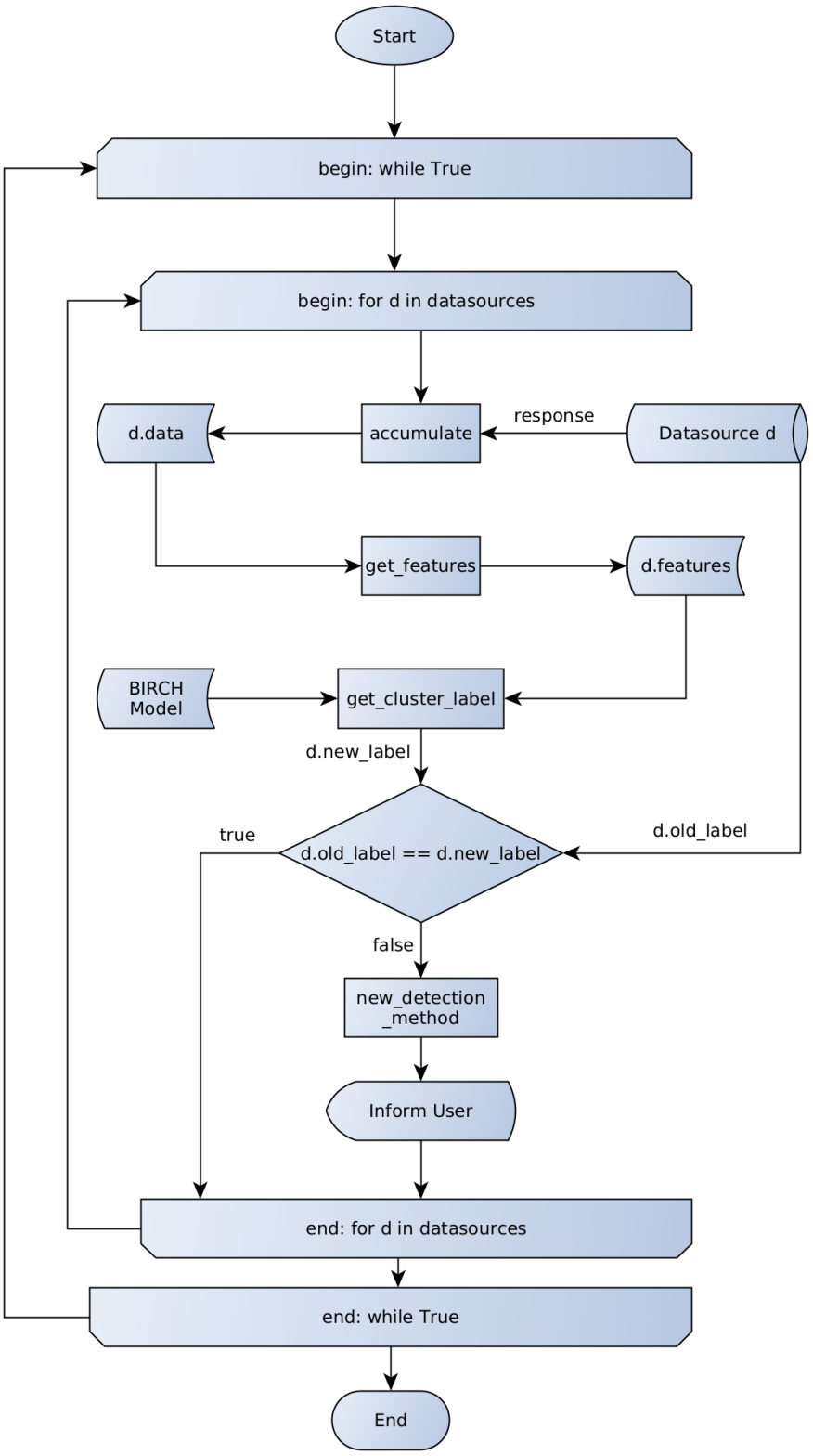

Figure 4.5: A flowchart showing the online clustering process.

with the *scikit-learn* scale function. Then the training function computes the score[3] of each value in the given dataset and returns a threshold regarding this score, which lies in a small quantile (0.5%) of all scores. Meaning, when detecting anomalies in the future, scores which are equal to or smaller than the scores in this quantile are considered anomalous.
The KDE detection function computes the scores of all values in a given time series data window and adds the indices of values with scores below the threshold to a list of anomalies.

Matrix Profile Matrix Profile does not need a training function. We use the *matrixprofile* library to compute the Matrix Profile (see Section 3.4.2) [BOBM20]. To do so, we have to provide the time series data and a size for the sliding window. We implement the sliding window to be dependent on the size of the time series. Once we obtain the Matrix Profile, we can use the *discover.discords* function of the matrixprofile library to find anomalous values in the data.

Long Short-Term Memory For LSTM we have to implement a training function to train the neural network. We use *tensorflow-keras* to build a small *Sequential* model consisting of two *LSTM* layers and a *Dense* layer [AAB+15]. Both *LSTM* layers use the tanh (see Appendix 2) activation function. The whole model is optimizing for the mean squared error (MSE, see Appendix 1) loss function and is training with sequences of length 5 from the normal time series data. This means we take a subsequence of length 5 from the data as a training sample and the single value directly following this subsequence as the expected label. We also use the *EarlyStopping* callback function when training the model, which allows the training to stop once the results of the loss function stop decreasing. If the training does not stop due to early stopping it will continue until it reaches 1300 training epochs.
We use a simple heuristic to determine whether an observed value is an anomaly. From the model training, we save the maximum result of the loss function. This maximum is compared to the mean squared error between the observed and predicted value. Predictions where the MSE is larger than the maximum MSE recorded during model training are considered anomalous.

[3]The score is the normalized result of the log-likelihood function. This function describes how likely it is for a sample to appear in a sample distribution. The function is also dependent on the parameters of the process that produces this distribution.

4.4 Data Configuration

We provide three different interfaces to configure possible data sources: Artificial data sources, file data sources and stream data sources. Artificial data sources can be configured to emit generated data and are mainly used for unit tests. File data sources enable us to load historic log data from CSV files. This provides the basis for offline training. Stream data sources can be configured to poll a HTTP (Hyper Text Transfer Protocol) endpoint for values. They can be used to accumulate data for offline training too when there is no historic log data available, but are mainly intended to be used in online anomaly detection. Each data source is a python object containing all information regarding a given time series.

4.4.1 File Data Sources

File data sources are used to read and preprocess data from a CSV file. Configuration parameters for such data sources include a *name* and a *filename* to identify the source internally and system wide. We also have to provide the *column*, where raw values can be found in multidimensional datasets (see Section 3.1.2). Some files might contain a header or footer with meta information on the dataset. These can be truncated by setting *head* and *tail* parameters in the configuration file. We also have to set the *delimiter* used to separate values in the file. Finally we can configure the data source to *subsample* the available data. If this parameter is set to an integer $k > 0$, the data source will only use every k samples to populate its internal data array.

4.4.2 Stream Data Sources

Similar to file data sources each stream data source has to have a *name* and a system wide identifier. This identifier is an *address* to a HTTP endpoint. How often this endpoint is polled for new values can be defined via the *frequency* parameter. It is a floating point value defining the number of seconds to wait between each request. Different emitters of time series data might supply different formats of endpoint data. We implemented the parameter *pattern* to identify the relevant parts of the submitted data. Suppose the received data has the form of *[42,14:52:34]*. The pattern might be defined as *[$value$,$hour$:$minute$:$second$]*, where the $ symbol indicates the start and the end of a variable. This way we can interpret the received data accordingly and extract the needed values. Additionally we define a *window* which states how many samples should be collected before analysing them. A boolean parameter *use_for_training* defines whether a *Stream data source* is used to collect training data from the stream endpoint. If set to *false*,

one has to define a corresponding *File data source* that provides historical log data for this *Stream data source*.

4.4.3 Example Configuration

In Listing 4.1 we show an example of a configuration file. This file contains two *File data sources* and two corresponding *Stream data sources* and no *Artificial data sources*. The configuration must be provided in JavaScript Object Notation (JSON) and is parsed by the prototype system at the initial system start and at some points during runtime. The *File data sources* in this example consist of two similar log files with historical time series data from the Numenta Anomaly Benchmark set (NAB see Section 5.3.2). We can see the configuration parameters described in the previous Sections (4.4.1 and 4.4.2).

Listing 4.1: Example configuration

```
{
"DataSources":  {
        "FileDataSources" :     [
                            {
                            "name" : "ec2_cpu_utilization_5f5533",
                            "filename" : "NAB_data_train\\
                                ec2_cpu_utilization_5f5533.csv",
                            "column" : 2,
                            "head" : 0,
                            "tail" : 0,
                            "delimiter" : ",",
                            "subsample" : 1
                            },
                            {
                            "name" : "ec2_cpu_utilization_24ae8d",
                            "filename" : "NAB_data_train\\
                                ec2_cpu_utilization_24ae8d.csv",
                            "column" : 2,
                            "head" : 0,
                            "tail" : 0,
                            "delimiter" : ",",
                            "subsample" : 1
                            }
        ]
        ,
        "StreamDataSources" :   [
                            {
                            "name" : "ec2_cpu_utilization_5f5533",
                            "address" : "http://127.0.0.1:3333/validation/
                                ec2_cpu_utilization_5f5533",
```

```
28                          "frequency" : 0.5,
29                          "pattern" : "{$name$: $value$}",
30                          "window" : 150,
31                          "use_for_training" : false
32                          },
33                          {
34                          "name" : "ec2_cpu_utilization_24ae8d",
35                          "address" : "http://127.0.0.1:3333/validation/
                                ec2_cpu_utilization_24ae8d",
36                          "frequency" : 0.5,
37                          "pattern" : "{$name$: $value$}",
38                          "window" : 150,
39                          "use_for_training" : false
40                          }
41          ]
42          ,
43          "ArtificialDataSources" :       []
44          }
45  }
```

After parsing this configuration the prototype system creates two *FileDataSource* and two *StreamDataSource* python objects which are passed around from module to module as the system runs. Initially the *FileDataSources* are populated with data from their respective CSV files. In this case we select the 2nd (starting at 0) column from the file. The CSV parser reads every single value from the given column, since we have configured *head* and *tail* as 0, meaning we do not truncate any data from the beginning or end of the file. We also set *subsample* to 1 resulting in the line counter increasing once for every line read. This concludes the data processing phase and the system enters the training phase (see 4.1). Once both *FileDataSources* have obtained a cluster label and a corresponding detection method the system enters the *Usage Phase*. At the beginning of this phase the previously initialized *StreamDataSources* are compared to the *FileDataSources*. In this case the two *StreamDataSources* match the names of the two *FileDataSuorces*. As the *StreamDataSources* are configured not to be used for training, they inherit the attributes from their corresponding *FileDataSources*. Inherited attributes are cluster labels, detection methods, original CSV data and, if present, evaluation results. With this information the *StreamDataSource* objects are now used to continuously request (*accumulate*) values from their respective endpoint addresses at the provided frequency. Our configuration shows that the endpoints provide values as JSON objects in the form *{$name$: $value$}*.

A valid response of the endpoint *http://127.0.0.1:3333/validation/ec2_cpu_utilization_24ae8d* might look like this:

```
{"cpu_util":2.6}
```

After receiving this response the value 2.6 will be appended to the *StreamDataSource*'s current detection window. This window is configured to the size of 150 and is initially filled with the last 150 data samples from the parsed CSV file. During the *Usage Phase*, the detection window behaves as a sliding window. While appending the value 2.6, the oldest value at the beginning of the window gets removed. This window is passed to the *Online Clustering* and the window is assigned a cluster label (see Section 4.2.3). Finally the detection method corresponding to the assigned cluster label is executed on the data in this window and the results are presented to the user.

5 Evaluation

In this chapter, we evaluate the developed system. First we describe the evaluation criteria and requirements that we would expect such a system to fulfil. Then we provide details on the datasets we use to configure and evaluate our system. After this, we explain the details of the methods employed for evaluation. Finally we present and discuss the results of our system evaluation.

5.1 Evaluation Criteria

We divide evaluation criteria into general and special criteria. The general criteria apply for all developed systems in the ICS context. Special criteria are dependent on the use case and components we explore in our approach. These criteria concern anomaly detection systems and our effort to reduce user interaction.

5.1.1 General Criteria

Usual requirements for newly installed industrial IT Security systems are:

- "No change in round trip time of network packets"[AFS17]
- "No interruption or manipulation of the productive network"[AFS17]
- "No reduction of overall network security"[AFS17]
- "Adaption capability for legacy devices"[AFS17]

Our system is intended to be a consumer of middleware data and not to forward this data to other consumers in the network. The middleware is independent of our system and currently consists of multiple endpoints that each provide single time series data points when requested. This implies that round trip time of network packets and interruption or manipulation of the productive network are no concern for our implementation. The influence on overall network security strongly depends on the host system for our application. The application itself currently does not provide any interface other than local (command line and file) output and a constrained configuration file. Communication with the middleware is currently neither encrypted nor verifiable, as it is polled via plaintext HTTP requests. A man-in-the-middle attack could undermine data confidentiality and

integrity. Adaption capability for legacy devices depends on the middleware once again, since it is responsible for passing device data to our application.

5.1.2 Special Criteria

Special criteria concern anomaly detection and classification problems. We first describe rules and methods of performance evaluation of anomaly detection and clustering. Then we explain important time related aspects of our evaluation effort.

Anomaly Detection based IDS Schneider et al. formulated general advice for intrusion detection systems based on anomaly detection [Sch19]. Two of their rules are especially applicable to our implementation:

1. "Don't mistake high recall values caused by high false-alarm rates for good performance."[Sch19]
2. "Don't misinterpret high detection capabilities under worst-case scenarios."[Sch19]

Worst-case scenario means a scenario where anomalies are very obvious (e.g. a total system failure). More nuanced outliers might worsen detection capabilities but are often overlooked in evaluations. We respect these two rules in our evaluation by contrasting recall values with the more balanced F1-Score and evaluating performance under realistic assumptions rather than worst-case scenarios.

Clustering Performance The goal of the system is to find the best anomaly detection method for any given data source without explicitly computing the detection performance indicators (see Section 3.6.2) of each detection method for each data source. We cluster similar data sources, compute detection performance indicators for each method on a single (representative) data source from each cluster and assume that the method with the best performance indicators for this data source performs equally well on similar data sources in the same cluster. To evaluate the performance of the system we explicitly compute the detection performance indicators of each detection method for each data source (brute force) and compare these methods with the ones selected by clustering.

Training Time There are two time components which we will evaluate. First the time the system needs to evaluate and select an anomaly detection method compared to the time an evaluation of all methods on all data sources via brute force would take. We call this the evaluation of *training time*. Second we will compare the execution times of the currently implemented anomaly detection methods. We call this the evaluation of *detection latency*.

Detection Latency In a real time industrial control system many components and their respective actions are time critical. Any disruption or anomalous behaviour should therefore be detected and mitigated as soon as possible. The speed of detection (detection latency) should be considered a mission critical parameter when evaluating intrusion detection or anomaly detection systems [MC14].
Although this parameter is so important it has to be balanced with another important parameter, the detection precision, as one of them increases in most cases the other decreases and vice versa. Usually this balance depends on the amount of analysed data. Less data leads to fast analysis but tends to produce unreliable results. Whereas more data (up to a certain point) makes results more reliable but also takes more time to analyse.
Our implementation addresses this trade-off by enabling the user to define the amount of analysed data at each time step in a configuration file (see Section 4.4.3). The *windowsize* parameter for *Stream Data Sources* denotes the number of samples to be analysed.

5.2 System Parameters

Our prototype system and its individual modules provide many options to configure parameters with an impact on module and system performance. We need to distinguish three types of parameters. *Data dependent* parameters vary in each use-case and installation. *Fixed* parameters are part of the system specification and do not get changed. *User defined* parameters are provided in a configuration file when setting up the system.

Data Dependent Parameters Some system parameters are dependent on the data that the system is executed on. We focus on data dependent parameters needed for our clustering procedures. We explore reasonable ranges for these parameter values regarding our available datasets in Section 5.4.1. There are multiple other parameters especially for neural network based detection methods like LSTM, which might be tweaked to improve performance depending on the input data. For the sake of simplicity regarding our prototype implementation we establish a fixed set of values for these parameters, although performance might be improved using different values for different input data.

Fixed Parameters Fixed parameters have an impact on the training phase of the system and should generally be independent of the available datasets and the system environment. We evaluate the impact on the training phase of the number of injected anomalies for our anomaly generator (see Section 5.5.1) and the sequence length of the input sequence for our LSTM anomaly detection (see Section 5.4.2).

User Defined Parameters User defined parameters can be set according to user preferences. As explained in Section 5.1.2, the detection latency has to be balanced with the detection precision. This trade-off is dependent on the window size, which is the only user defined parameter we evaluate. Different window sizes and their impact on detection latency are evaluated in Section 5.5.3.

5.3 Evaluation Datasets

In the following section we describe the datasets we use for evaluating and exploring the different parameters effects on single components of the system. Then we give a similar description of the datasets we use to evaluate the finished prototype system.

5.3.1 Parameter Selection Dataset

To establish initial values for the three different types of parameters, we use a dataset of recorded energy consumption values from the *tracebase* repository [RBB+12]. This dataset contains time series data of energy consumption of different electronic devices (freezer, desktop PC, lamp, etc.). We use this dataset to feed different submodules of our implementation and get a general idea of reasonable parameter values and their impact on the respective submodules' performance. The time series data in this repository is not labelled and may contain anomalies. For our selected datasets, we assume there are no anomalies (although there might be some) as we would during initial deployment of our system.

5.3.2 System Evaluation Dataset

To eliminate any potential source of bias in our system evaluation, we use an entirely different dataset to conduct the final evaluation in Section 5.5.1. The Numenta Anomaly Benchmark set (NAB) contains multiple time series of different origins (advertisement clicks, server metrics, freeway traffic, etc.) [ALPA17]. We use a subset of this data, the server metrics, which contains data on CPU (Central Processing Unit) utilization of a computer in a data centre. Since this data contains labelled anomalies, we can compare the anomalies found by the system to the ones actually present and thereby evaluate our prototype system.

5.4 Parameter Selection

Before evaluating the final prototype system, we have to establish reasonable initial configuration parameters for our system and its single modules. Using the *tracebase* dataset we first explore *data dependent* parameters for the clustering algorithms. We also establish

fixed parameters for our prototype system. We find these parameters using a grid-search. A multidimensional list (*grid*) of possible parameter value configurations is used to build and train different models. These models are evaluated and ranked by their respective performance. After finishing the grid-search the parameter value configuration leading to the best performance is selected for further usage.

5.4.1 Data Dependent Parameter Exploration

Ideally, a truly autonomous system would be able to infer *data dependent* parameters from the given data. Our implementation might be headed in this direction but is currently not able to configure these parameters autonomously. We therefore have to try numerous configurations via grid-search to find suitable parameter values.

Clustering Parameters We introduced two different unsupervised clustering algorithms in Chapter 3. In the following paragraphs we explore which parameter values are needed for them to perform well on the *tracebase* dataset and provide reasons on why BIRCH is a better fit for our use case than DBSCAN.

DBSCAN Parameters The DBSCAN clustering algorithm requires the two parameters *eps* and T_{MinPts} defining the Eps-Neighbourhood and the minimum number of points required to start a cluster respectively (see Section 3.5.2). For the *eps* parameter we provide the values 0.5, 2, 30 and 500. The T_{MinPts} parameter has the value 5. To create data points for DBSCAN to cluster, we select 18 *tracebase* datasets and compute their features (see Appendix Listing 1). Half of these datasets are generated by one process (energy consumption of a freezer) and the other half is generated by a second process (energy consumption of a network switch). The clustering algorithm should be configured to group the computed features of these datasets with regards to their generating processes. A meaningful visual representation of the multidimensional feature space can be given by using Principal Component Analysis[4] (PCA) to reduce the feature dimensions to two.
Figure 5.1 shows the target feature clusters on the left (a)) with the first principal component on the x axis and the second principal component on the y axis. The target clusters are labelled (coloured) according to their respective datasets. The right side (b)) shows the clusters obtained by the DBSCAN algorithm executed with default parameters ($eps = 0.5$, $T_{MinPts} = 5$) on the extracted features. Even without any data dependent configuration

[4]Principal Component Analysis is a technique to reduce the dimensionality of data. The general idea is to represent the data using the dimensions where the data distribution has the highest dispersion (e.g. has the greatest variance), since these dimensions are assumed to carry the most information on differences between single data points [Agg15].

parameters DBSCAN delivers a good estimate of the number of expected clusters and the feature cluster labels.

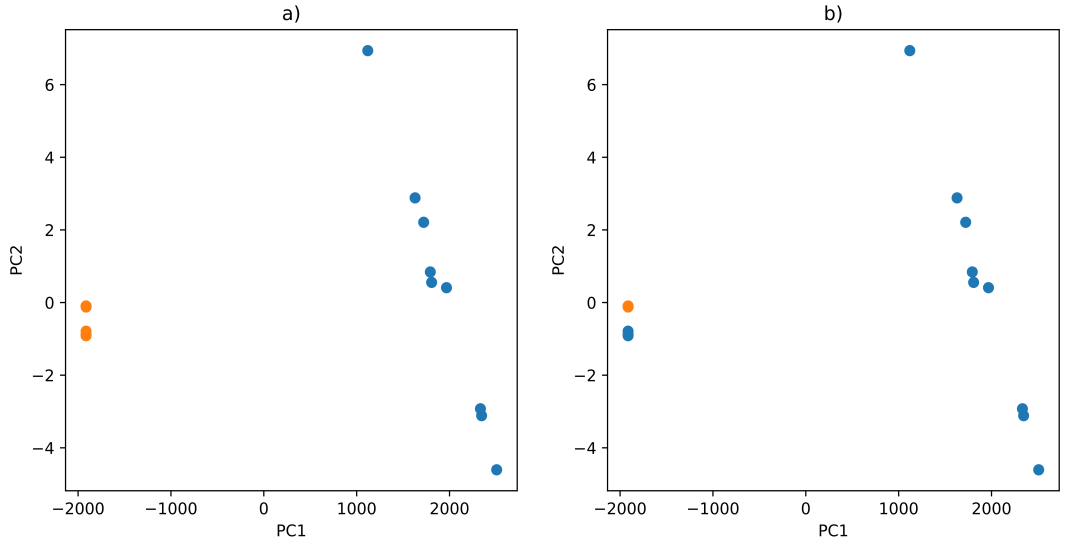

Figure 5.1: Two target feature clusters (orange and blue) on the left (a), compared to default DBSCAN output on the right (b). PC1 and PC2 are the first two principal components resulting from the PCA.

Figure 5.2 illustrates clustering results of DBSCAN configured with different values (2, 30, 500) for *eps*. The *eps* values of 2 and 500 result in clusterings that do not match our target clustering in Figure 5.1 a). Only the *eps* value 30 successfully produces the desired clustering of the given datasets. We assume that an *eps* value of 30 is a good choice when performing clusterings on similar data in the future.

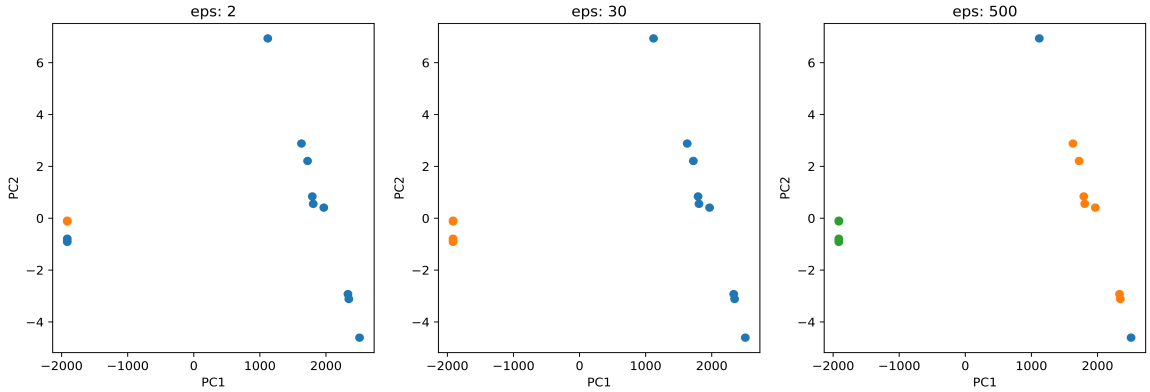

Figure 5.2: Three images showing DBSCAN clustering results for values 2, 30 and 500 for the *eps* parameter.

As mentioned in Chapter 3, DBSCAN does not offer a way to incrementally insert new data points (feature vectors of datasets) once the clustering is complete. Adding new data sources while the system is running would trigger a new clustering for all available data sources and delay detections based on this new clustering substantially. The *sklearn* implementation of DBSCAN also operates on batch data, meaning we are unable to request a label for a single feature vector. This would make a new clustering for all feature vectors necessary each time a new feature vector is computed during live anomaly detection. These limitations lead to the conclusion that DBSCAN is not suitable for our use case.

BIRCH Parameters As presented in Section 3.5.3 the BIRCH tree data structure requires two parameters. The threshold T is the maximum diameter of a subcluster in a leaf node. The branching factor B determines the maximum number of children of a single node. Similar to the previous grid-search we define multiple values for T (0.5, 50, 450 and 3000) and a fixed value (50) for B. We use the same 18 datasets and features used for the DBSCAN grid-search and use BIRCH to compute a clustering of these same features.
In Figure 5.3 subfigure b) we see that nearly every single feature vector is labelled as a separate cluster when using BIRCH with default configuration parameters $T = 0.5$ and $B = 50$.

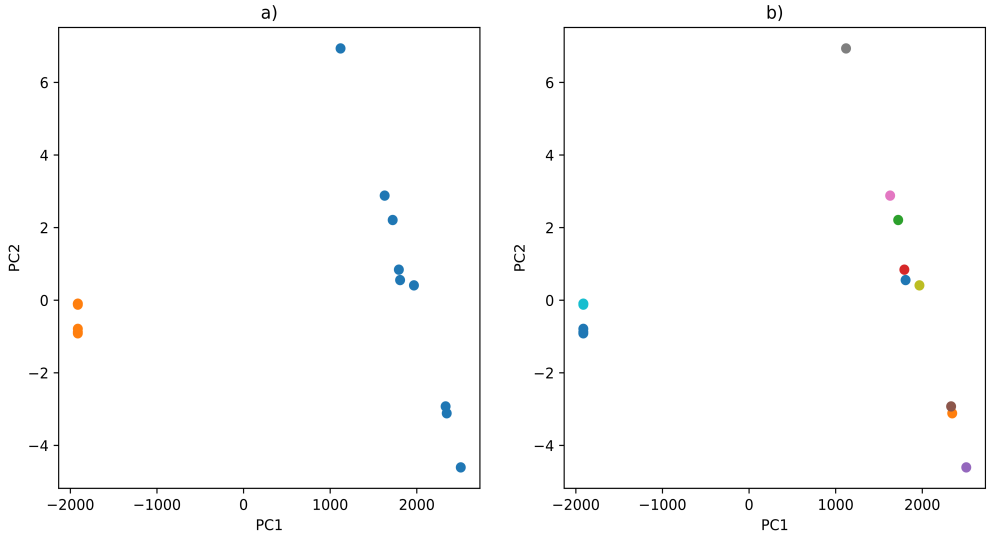

Figure 5.3: Two target data source clusters (a) compared to BIRCH feature clustering with default parameters (b).

To obtain results closer to our target clustering (subfigure a)) we have to pass a higher values for parameter T to BIRCH. Figure 5.4 shows a comparison of different threshold values. At a threshold of 50 the BIRCH algorithm correctly identifies the left most cluster

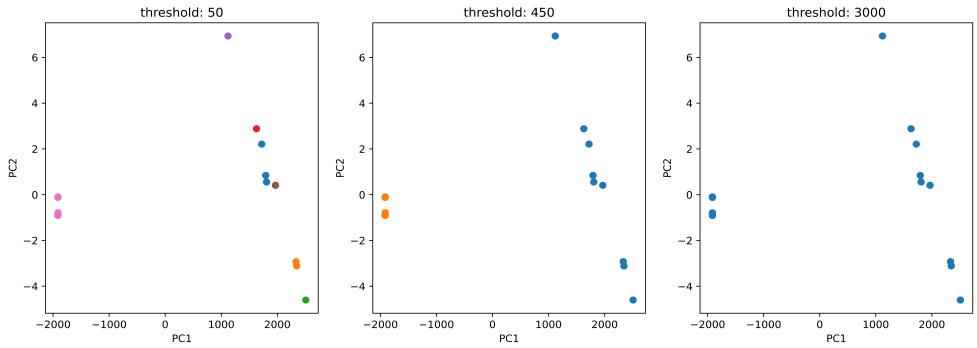

Figure 5.4: Three images showing BIRCH clustering results for values 50, 450 and 3000 for the *threshold* parameter.

but still struggles to group the points on the right side. An increased threshold value of 450 leads to an exact match of the clustering result with the target clustering. This indicates that 450 is a good threshold for the used datasets and their corresponding features. At a threshold of 3000 the BIRCH algorithm stops providing useful clusterings and labels every dataset the same. We configure the BIRCH clustering with a threshold value of $T = 450$ and a branching factor $B = 50$ for our subsequent parameter selections based on the *tracebase* data, assuming that the clustering will behave similarly on similar datasets. The BIRCH clustering data structure is suitable for our use case as it supports iterative insertions of new data points as well as classification of single data points without triggering a whole new clustering process for each new data point.

5.4.2 Fixed Parameter Selection

Using a grid-search similar to the ones used for DBSCAN and BIRCH we explore different input sequence lengths and activation functions in our LSTM neural network to see their impact on anomaly detection performance. The prototype implementation contains many more of such fixed parameters and this elaborate presentation of the parameter selection process is representative for similar approaches used to find these other parameter values.

LSTM Sequence Length We explore different input sequence lengths and activation functions for the LSTM neural network. The input sequence length determines how many single samples are used as an input for the model to predict the value of the subsequent sample. We examine sequence lengths ranging from 5 to 45 in steps of 5. The second parameter used in our grid-search is the activation function. This activation function determines the output produced by a single neuron. We evaluate the impact of the three most common activation functions *relu*, *tanh* and *sigmoid* (see Appendix 2) on the LSTM

anomaly detection performance. We use four different datasets from the *tracebase* repository and train our LSTM model on this data. After randomly injecting artificial anomalies we let the model detect these anomalies and we compute the evaluation metrics precision, recall, F1-Score and accuracy. We train and evaluate the model ten times for each input length on each dataset to mitigate the effects of the random initial LSTM weights.

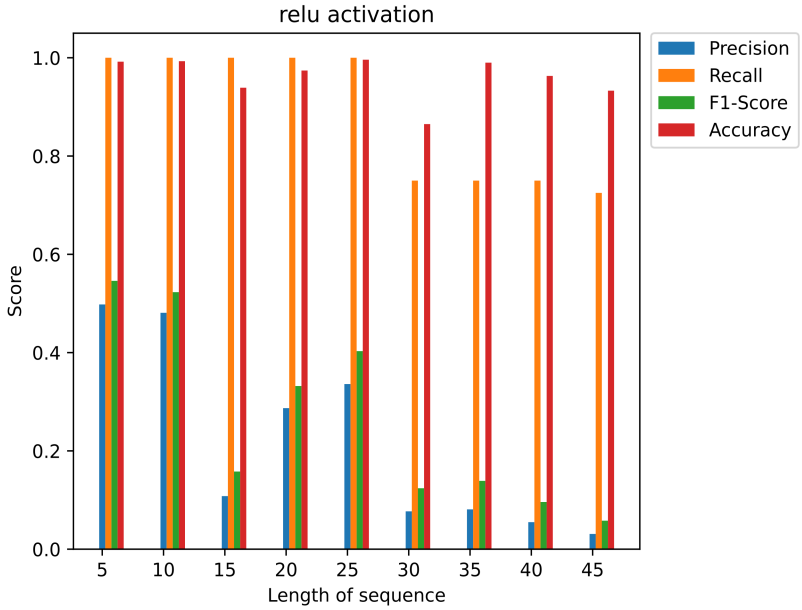

Figure 5.5: Different evaluation scores are achieved by LSTM detection depending on the size of the input sequence. Executed with the ReLU activation function. See Appendix Table 4.

Figure 5.5 shows that results from LSTM anomaly detection depend heavily on the selection of the input sequence length. In addition to a faster training and detection time the models with smaller input lengths appear to perform better than models with greater input lengths. An input length of 5 produces the best average F1-Score, as can also be seen in Appendix Table 4. An exemplary illustration of a single detection run with different sequence lengths can be found in the Appendix Figure 4.

Figure 5.6 shows a similar trend of worsening detection capabilities with rising sequence lengths. It also shows that the model based on the tanh activation function generally performs better than the ReLU based model.

Appendix Table 5 as well as Figure 5.7 show results for a model base on the sigmoid activation function. Based on these results, we select a constant input sequence length of 5 for our LSTM model in combination with the *tanh* activation function.

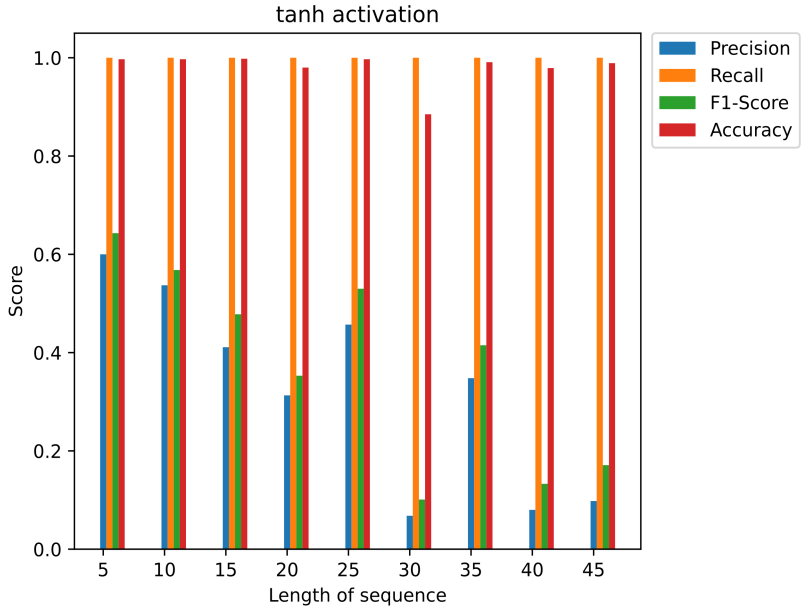

Figure 5.6: Different evaluation scores are achieved by LSTM detection depending on the size of the input sequence. Executed with the tanh activation function. See Appendix Table 3.

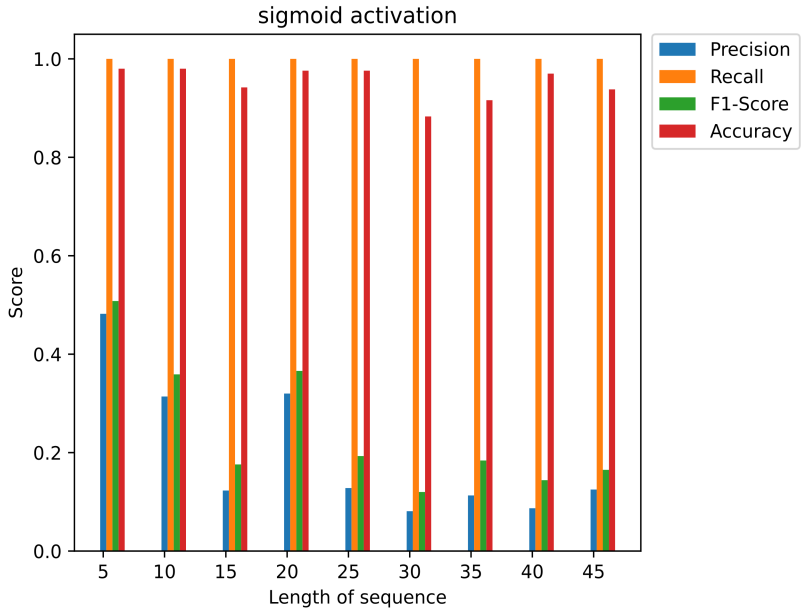

Figure 5.7: Different evaluation scores are achieved by LSTM detection depending on the size of the input sequence. Executed with the sigmoid activation function. See Appendix Table 5.

5.5 System Evaluation

In this section we present evaluation results relevant to our research questions stated in Chapter 1. We want to explore ways to evaluate anomaly detection methods on unlabelled data. We also intend to find a way to efficiently select anomaly detection methods for a large amount of data sources. In the following sections we describe how well the implemented solutions for these problems perform in comparison to simple brute force solutions.

5.5.1 Anomaly Generation Performance

In this evaluation we assess how well our anomaly generation and subsequent method selection can predict the actual best anomaly detection method. We are using four NAB datasets as log data of four different data sources. Pairs of the dataset ID and the respective NAB dataset name are noted in Appendix Table 9. These datasets contain percentage values of CPU utilization of a computer sampled every five minutes. The datasets also contain one known anomaly each. We split each dataset into training and evaluation data. The training data of Datasets 1 and 3 contain 999 samples each. The training data of Dataset 2 contains 3299 samples and that of Dataset 4 contains 1499 samples. These training datasets do not contain any known anomalies and are used to build groups of similar data sources and select the best anomaly detection method during the training phase (see Figure 4.1).

The datasets are similar enough for our clustering to group them into a single cluster.

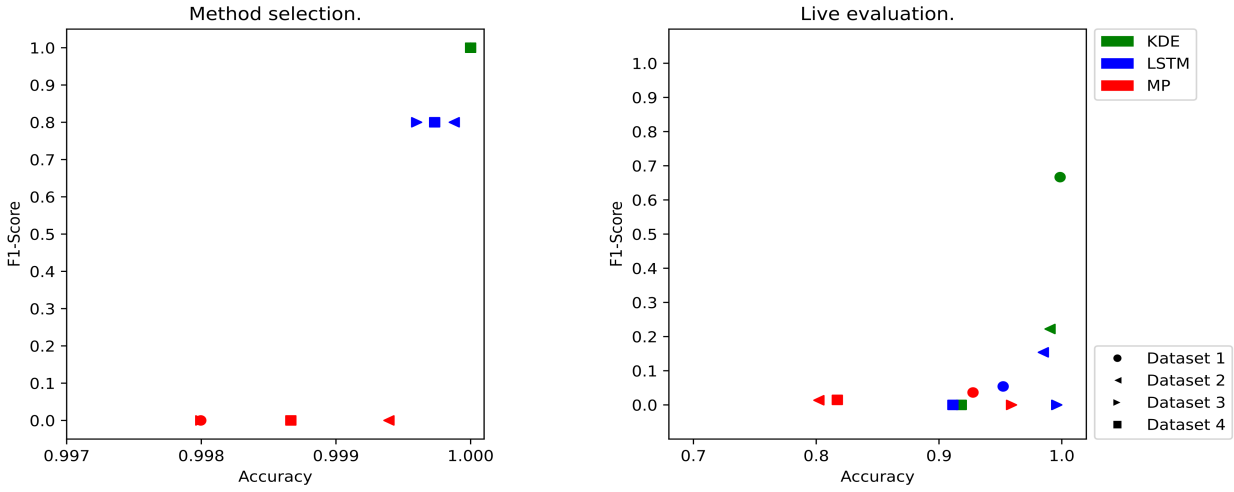

Figure 5.8: Evaluation results of offline method selection during training compared to online live anomaly detection. See Appendix Table 9.

This leads to a single best detection method selected for all of the available datasets. In this case KDE outperforms LSTM and MP detection methods during method selection. This can be seen in Figure 5.8 in the method selection (left) subfigure. The left subfigure shows the results of a brute force evaluation of all detection methods on all datasets. The y axis represents F1-Scores and the x axis the accuracy resulting from this brute force evaluation. The top right corner of the plot is the best possible evaluation result with an F1-Score and Accuracy of 1 respectively. In Appendix Table 9 we see that during training the KDE detection method achieves perfect evaluation results on all datasets. The values are averages over five evaluation runs during the training phase. In each run a single anomaly is randomly injected into the training data and the detection methods are used to find them. The average LSTM F1-Score of 0.8 for Dataset 2 indicates that the LSTM detection method found the injected anomaly four out of five times, while the average F1-Score of 1.0 for Dataset 1 indicates that the LSTM detection method found the injected anomaly each time. According to this offline evaluation the KDE detection method should be selected for live anomaly detection because it will perform better than the alternative methods. When running the application this offline evaluation is only executed for one of the datasets since the datasets all belong to the same cluster. The other dataset evaluations are just provided for this example.
The live evaluation results (right) in Figure 5.8 confirm the assumption that KDE will outperform the other methods. To conduct this live evaluation we served the evaluation data (containing realistic and known anomalies) via HTTP and let our prototype execute the anomaly detection on incoming data windows (see Figure 4.1). We executed this live detection for each detection method separately to create comparable results. In a real usage scenario only the KDE detection method as chosen during method selection would have been executed. For Dataset 1, 2 and 3 KDE achieves higher or equal F1-Scores and accuracy values than any of the other detection methods. It also dominates with regards to accuracy when evaluated on Dataset 4, but MP achieves a F1-Score higher than KDE on this particular dataset. We can see that the KDE and LSTM detection methods found the anomalies in Datasets 1 and 2 with high accuracy and have an overall higher accuracy than the MP detection method. This trend of the MP detection method's lower accuracy is already visible during method selection, although at a much smaller scale (see x axis scaling).
In conclusion we can see that during method selection the detection methods perform similar on similar datasets. The relative performance of these detection methods (method A achieves higher values in F1-Scores and accuracy than method B) is confirmed during live usage. Injecting the doubled maximum value of a training dataset once as a controlled anomaly enables us to roughly predict the relative performance of multiple anomaly de-

tection methods and select the detection method which presumably performs best.

Injected Anomalies When injecting anomalies into the normal training data there are several parameters that can be configured. The first parameter is the number of injected point anomalies. The number of anomalies should be low to match the anomaly definition given in Section 3.2. High numbers of anomalies in the training data would lead to false or unrealistic assumptions on the prevalence of anomalies in real data. The performance of the selected methods might not be comparable to the performance of methods used during live detection. Therefore we inject just a single anomaly during anomaly generation. Another configurable parameter is the scale of the injected point anomalies. That means we configure by how much the injected anomaly deviates from the normal and expected values of the training data. The impact of different anomaly scales was evaluated. We used different values as factors for multiplication with the maximum value during anomaly generation using Algorithm 2. The default value for this parameter is 2 (hence *double*) and we replaced this by 1.5, 1.2 and 0 without seeing changes in relative performance of the detection methods. Another question is the impact of doubled sequences (see Algorithm 3) on relative detection method performance. None of the three available detection methods found a doubled sequence in our four training datasets to be anomalous, which makes sorting and selection of detection methods by relative performance impossible. A doubled sequence might be considered anomalous in a dataset with clean, regular pattern frequency (e.g. sine). In this case the doubled sequence could be replaced by two point anomalies at the beginning and end of the sequence.

5.5.2 Training Time Evaluation

In this section we evaluate the time needed to train and evaluate detection methods. We compare a brute force approach against our implemented clustering and representative data source selection. The brute force approach evaluates every single anomaly detection method for every available data source. The selection by (offline) clustering and representative data source selection is explained in Section 4.1. We use 11 datasets from the *tracebase* repository and one not publicly available dataset of sea level recordings. The first of the 11 *tracebase* datasets contains energy consumption data of a freezer. The other 10 contain energy consumption data of a desktop computer for the duration of a single day each. The sea level data is a long record of hourly measurements for the duration of several years. Using these very heterogeneous datasets we provoke the system to produce multiple different clusters of data sources. We begin by training the system on two datasets and measure the time needed for the two solutions (brute force and representative selection) to compute their respective results. A result consists of a mapping of data

sources to their respective anomaly detection methods. Then we add a third dataset and start the training and selection process again. We repeat this for all 12 datasets. The execution times are measured on *Computer-1* with specifications listed in Appendix Table 1. An anticipated effect of the selection of representative data sources is the significant reduction in execution time when evaluating detection methods. When using brute force we evaluate each method for each data source separately. This leads to a linear increase in execution time in correlation with the number of data sources.
As can be seen in Figure 5.9, selecting a representative dataset for each cluster correlates the execution time with the number of clusters which is expected to be much smaller than the number of datasets.
The selected detection methods are the same in both cases (brute force and representative). Meaning the method selection via clustering yielded the same best methods for each dataset as the method selected by brute force. This could be due to our limited method pool only containing three detection methods to choose from and requires further evaluation as stated in Section 6.2.

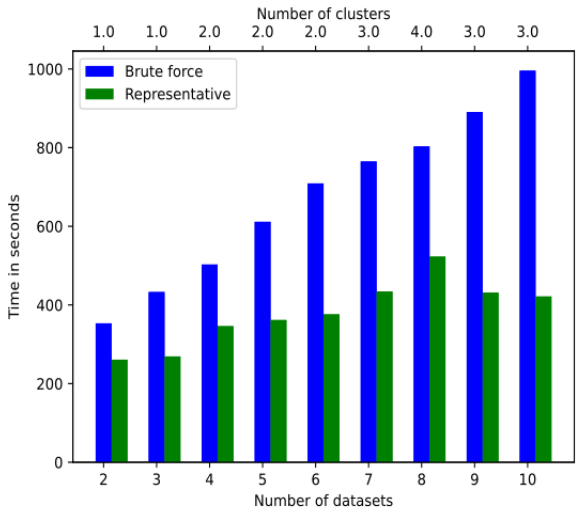

Figure 5.9: Execution time when evaluating methods in relation to the number of available datasets and the number of found clusters. See Appendix Table 6.

5.5.3 Detection Latency

In this section we measure how fast the different anomaly detection methods are able to deliver results. We use two NAB datasets in a similar configuration to the one used in Section 5.5.1. We force the system to select a detection method as a fixed parameter of the

evaluated configuration. Then we request new samples from the respective data sources and execute the selected method to find anomalies in the provided window. We request 10 samples and compute the mean execution time for the detection method over the 10 resulting data windows. This process is repeated for window sizes 50, 150, 250, 350, 450, 550 and 650. The execution times are measured on *Computer-2* with specifications listed in Appendix Table 2. As mentioned in section 5.1.2, the *windowsize* defines the amount of analysed data for each time step. We would expect this parameter to directly influence the amount of time needed for every single execution of a given anomaly detection method.

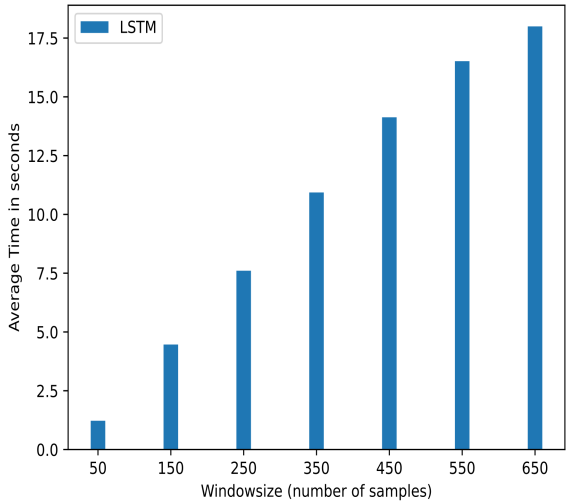

Figure 5.10: Execution time of LSTM anomaly detection in relation to the configured window size. See Appendix Table 7.

Figure 5.10 shows the average LSTM detection execution time over 10 requests in relation to the configured *windowsize*. As mentioned in Section 4.4.3, the window gets updated at each request and the selected anomaly detection method is executed on the window data. We can see a linear correlation of *windowsize* and execution time for the LSTM anomaly detection method.

Figure 5.11 shows a similar relation for the Kernel Density Estimation based detection method. We also see that the Matrix Profile discord detection execution time is not influenced by these small changes of *windowsize*. The difference is detectable when *windowsizes* differ in several orders of magnitude (e.g. 200 and 200000). The statistical methods (KDE and MP) are executed significantly faster than the LSTM neural network, as can be seen

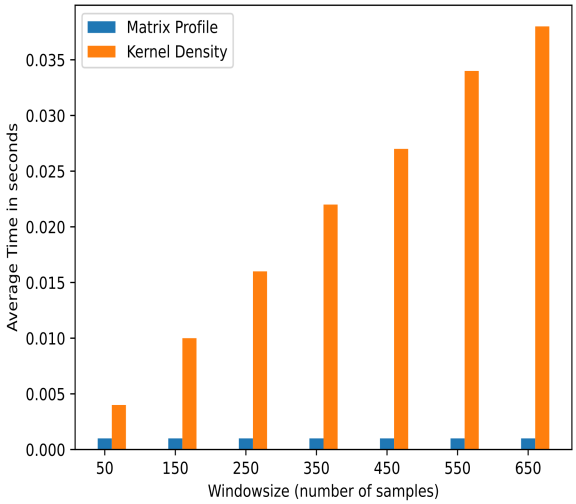

Figure 5.11: Execution time of Kernel Density Estimation anomaly detection and Matrix Profile Discord detection in relation to the configured window size. See Appendix Table 7.

when comparing the average time in seconds of both figures. These time measurements prove that execution times of different detection methods can vary significantly and are possibly correlated to the chosen window size. Not only the detection precision or F1-Score but also the execution time should be considered when selecting detection methods.

6 Conclusions

In this chapter we summarize the thesis and discuss the results as well as open problems and related questions.

6.1 Summary

In this thesis we designed, implemented and evaluated a system that clusters different data sources and applies anomaly detection methods on their emitted data. From each cluster, a representative data source is selected for thorough evaluation. During this evaluation, the system injects artificial anomalies into training data collected from the representative data source and runs every available anomaly detection method on this data to find the injected anomalies. The method with the best detection performance on the representative training data is then selected as the detection method for the whole represented cluster and will be applied to newly arriving data at runtime to find anomalies in real data. The detection methods available in the implemented method pool are based on *Kernel Density Estimation*, *Matrix Profiles* and *Long Short-Term Memory* neural networks.
Using this prototype system we explored the following research questions:

1. How can we evaluate anomaly detection methods on unlabelled data? As seen in Section 5.5.1, the problem of evaluating detection methods on unlabelled data can not be solved easily. Anomalies must be seen in a context of their surrounding data. Injecting single outliers at random locations often does not simulate a realistic scenario.

2. How can we select anomaly detection methods for a large number of data sources? Selecting anomaly detection methods for representatives of data source clusters appears to be a viable approach. Our evaluation in Section 5.5.2 shows that compared to a brute force approach a reduction in time needed for detection method selection is possible. With larger amounts of homogenous data sources this effect gets amplified.

3. How can we efficiently measure similarity of input datasets? While a supervised data source classifier as used by Reinhardt et al. might perform with high accuracy, a great amount of expertise and preliminary preparation is necessary to achieve this high accuracy [RBB+12]. Since we want to reduce the required level of expert knowledge and

preparation time we are using a fixed set of simple features and unsupervised classification (clustering) methods. Although this approach reduces preparation time prior to anomaly detection it can also reduce the clustering accuracy. An inaccurate clustering in our system leads to bad performance of the selected anomaly detection method (see Section 5.5.1). We also found that the BIRCH clustering data structure has advantageous attributes regarding our use case, such as iterative clustering shown in Sections 3.5.3 and 5.4.1.

6.2 Open Problems

The evaluation and subsequent conclusions show that our current prototype reduces the required user interaction by sacrificing anomaly detection performance. While the reduced user interaction might be a step in the right direction, the performance of anomaly detection methods must be a focus of future work. Especially in the context of intrusion detection systems. A failure to detect an intruder might result in irreparable damages to a company. The following list of open problems contains possible approaches to improve the current anomaly detection performance and proposes future improvements to the general system.

- **Anomaly generation**
  If we further want to follow the approach of generating artificial anomalies to evaluate the available detection methods we have to pose the question, how can we generate realistic artificial anomalous data and what is *realistic* in the first place? Our random anomaly injection approach was not sufficiently realistic. For an accurate detection method selection, we need to generate anomalies depending on the data source. An initial labelling of realistic anomaly values could be done manually by a user of the system. Although this approach could improve the method selection accuracy it requires at least some amount of expert knowledge and a significant amount of time. Ideally we would be able to find a heuristic to automatically create realistic anomalies.

- **Automated hyperparameter selection**
  Grid searches as conducted in Chapter 5 can be automated to find data dependent hyperparameters for clustering and anomaly detection methods during the learning phase of the system. This automation would further reduce the level of required expert knowledge and might additionally increase the system performance.

- **Clustering improvement**
  Currently the representative data source of a cluster is the first element of a gener-

ated cluster element list. To better represent a group of data sources the selected representative data source might be the one with the smallest distance to the cluster centroid instead.

- **Multiple dimensions**
  Some anomalies might only manifest in multiple dimensions and be otherwise invisible. We should not only analyse each data source on its own but also combinations of different data sources. There is interesting work by Zhao et al. [ZRA20] in this area that could be useful for this approach.

- **Further detection methods**
  With *Matrix Profiles* and *Long Short-Term Memory* neural networks we evaluated relatively new anomaly detection methods. To increase overall anomaly detection performance we might diversify the method pool by adding established detection methods such as *Isolation Forest* and *Local Outlier Factor*.

Regarding the implementation we also see possible improvements in the middleware communication and feature computation. Currently data is sent via HTTP. In a production environment some form of encryption should be implemented to ensure confidentiality and integrity of time series data. The *tsfresh* library is currently not optimized for streaming data. For more efficient computations we have to find another library or edit *tsfresh* to fit our needs. We also might expand the set of computed features and evaluate their relevance using *tsfresh* built in tools or other means (e.g. PCA). Regarding the evaluation of detection methods it would be an interesting approach to incorporate the method's execution time into the performance metric. To further evaluate the systems applicability to the real-world problem of intrusion detection, we might use test data which contains labelled attack intervals in an Internet of Things (IoT) context [VCA+20].

Nomenclature

ADS Anomaly Detection System, page 1

AE Autoencoder, page 3

API Application Programming Interface, page 3

FNR False Negative Rate, page 3

FPR False Prositive Rate, page 3

ICS Industrial Control System, page 1

IDS Intrusion Detection System, page 1

IoT Internet of Things, page 21

LSTM Long Short-Term Memory, page 2

MSE Mean Squared Error, page 17

REST Representational State Transfer, page 3

ROC Receiver Operating Characteristic, page 12

ROC-AUC ROC Area under Curve, page 12

SARIMA Seasonal Autoregressive Integrated Moving Average, page 2

SCADA Supervisory Control and Data Acquisition, page 8

TCP Transmission Control Protocol, page 3

References

[AAB+15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[AFS17] Simon Duque Anton, Daniel Fraunholz, and Hans Dieter Schotten. Angriffserkennung für industrielle netzwerke innerhalb des projektes IUNO. *arXiv*, 2017.

[Agg15] Charu C. Aggarwal. *Data Classification*. 2015.

[AKFS18] Simon Duque Anton, Suneetha Kanoor, Daniel Fraunholz, and Hans Dieter Schotten. Evaluation of machine learning-based anomaly detection algorithms on an industrial modbus/TCP data set. *ACM International Conference Proceeding Series*, 2018.

[ALPA17] Subutai Ahmad, Alexander Lavin, Scott Purdy, and Zuha Agha. Unsupervised real-time anomaly detection for streaming data. *Neurocomputing*, 262:134–147, 2017.

[AMK+20] Javed Asharf, Nour Moustafa, Hasnat Khurshid, Essam Debie, and Waqas Haider. A Review of Intrusion Detection Systems Using Machine and Deep Learning in Internet of Things : Challenges , Solutions and Future Directions. 2020.

[BOBM20] Andrew Van Benschoten, Austin Ouyang, Francisco Bischoff, and Tyler Marrs. Mpa: a novel cross-language api for time series analysis. *Journal of Open Source Software*, 5(49):2179, 2020.

[CBK09] Varun Chandola, ARINDAM BANERJEE, and VIPIN KUMAR. Anomaly Detection : A Survey. *ACM Computing Survey (CSUR)*, 41(3):1–72, 2009.

[CBNKL18] Maximilian Christ, Nils Braun, Julius Neuffer, and Andreas W. Kempa-Liehr. Time Series FeatuRe Extraction on basis of Scalable Hypothesis tests (tsfresh – A Python package). *Neurocomputing*, 307:72–77, 2018.

[CC19] Raghavendra Chalapathy and Sanjay Chawla. Deep learning for anomaly detection: A survey. *arXiv*, pages 1–50, 2019.

[DAFS19] Simon Duque Anton, Lia Ahrens, Daniel Fraunholz, and Hans Dieter Schotten. Time is of the essence: Machine learning-based intrusion detection in industrial time series data. *IEEE International Conference on Data Mining Workshops, ICDMW*, 2018-Novem:1–6, 2019.

[Dan02] Andrew W. Moore Dan Pelleg. X-means: Extending K-means with Efficient Estimation of the Number of Clusters. *ICML '00: Proceedings of the Seventeenth International Conference on Machine Learning*, pages 727–734, 2002.

[DBY+19] J. Dinal Herath, Changxin Bai, Guanhua Yan, Ping Yang, and Shiyong Lu. RAMP: Real-Time Anomaly Detection in Scientific Workflows. *Proceedings - 2019 IEEE International Conference on Big Data, Big Data 2019*, pages 1367–1374, 2019.

[DFM+06] Holger Dreger, Anja Feldmann, Michael Mai, Vern Paxson, and Robin Sommer. Dynamic application-layer protocol analysis for network intrusion detection. *15th USENIX Security Symposium*, pages 257–272, 2006.

[EKSX96] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. *Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining*, pages 226–231, 1996.

[FCR19] Robert Flosbach, Justyna Joanna Chromik, and Anne Remke. Architecture and prototype implementation for process-aware intrusion detection in electrical grids. *Proceedings of the IEEE Symposium on Reliable Distributed Systems*, (October 2018):42–51, 2019.

[GBB11] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep Sparse Rectifier Neural Networks. *Proceedings of the 14th International Con- ference on Artificial Intelligence and Statistics (AISTATS)*, 15:315–323, 2011.

[HCH19] Ruei Jie Hsieh, Jerry Chou, and Chih Hsiang Ho. Unsupervised online anomaly detection on multivariate sensing time series data for smart manufacturing. *Proceedings - 2019 IEEE 12th Conference on Service-Oriented Computing and Applications, SOCA 2019*, pages 90–97, 2019.

[HCW20] Yuanduo He, Xu Chu, and Yasha Wang. Neighbor profile: Bagging nearest neighbors for unsupervised time series mining. *Proceedings - International Conference on Data Engineering*, 2020-April:373–384, 2020.

[KG09] Petr Kadlec and Bogdan Gabrys. Architecture for development of adaptive on-line prediction models. *Memetic Computing*, 1(4):241–269, 2009.

[KUF+20] Boo Joong Kang, David Umsonst, Mario Faschang, Christian Seitl, Ivo Friedberg, Friederich Kupzog, Henrik Sandberg, and Kieran McLaughlin. Intrusion resilience for PV inverters in a distribution grid use-case featuring dynamic voltage control. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 11777 LNCS:97–109, 2020.

[LASE18] Rocio Lopez Perez, Florian Adamsky, Ridha Soua, and Thomas Engel. Machine Learning for Reliable Network Attack Detection in SCADA Systems. *Proceedings - 17th IEEE International Conference on Trust, Security and Privacy in Computing and Communications and 12th IEEE International Conference on Big Data Science and Engineering, Trustcom/BigDataSE 2018*, pages 633–638, 2018.

[MC14] Robert Mitchell and Ing Ray Chen. A survey of intrusion detection techniques for cyber-physical systems. *ACM Computing Surveys*, 46(4), 2014.

[MK21] Abdullah Mueen and Eamonn Keogh. Time series data mining using matrix profiling: A unifying view of motif discovery, anomaly detection, segmentation, classification, and similarity joins. `https://www.cs.ucr.edu/~eamonn/Matrix_Profile_Tutorial_Part2.pdf`, 2017 [Online; accessed 01-October-2021]. ACM 23rd SIGKDD Conference on Knowledge Discovery and Data Mining.

[MLS18] Stefan Marschalek, Robert Luh, and Sebastian Schrittwieser. Endpoint Data Classification Using Markov Chains. *Proceedings - 2017 International Conference on Software Security and Assurance, ICSSA 2017*, pages 56–59, 2018.

[MVSA15] Pankaj Malhotra, Lovekesh Vig, Gautam Shroff, and Puneet Agarwal. Long Short Term Memory Networks for Anomaly Detection in Time Series.

ESANN 2015 proceedings, 20th European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning, (April):89–94, 2015.

[MWK+06] Ingo Mierswa, Michael Wurst, Ralf Klinkenberg, Martin Scholz, and Timm Euler. Yale: Rapid Prototyping for Complex Data Mining Tasks Ingo. (August 2006):935, 2006.

[Ola21] Christopher Olah. Understanding lstm networks. `https://web.archive.org/web/20211012105621/http://colah.github.io/posts/2015-08-Understanding-LSTMs/`, 2015 [Online; accessed 13-October-2021].

[PBDM20] Tom Puech, Matthieu Boussard, Anthony D'Amato, and Gaëtan Millerand. A Fully Automated Periodicity Detection in Time Series. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 11986 LNAI:43–54, 2020.

[PVG+11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[RBB+12] Andreas Reinhardt, Paul Baumann, Daniel Burgstahler, Matthias Hollick, Hristo Chonov, Marc Werner, and Ralf Steinmetz. On the accuracy of appliance identification based on distributed load metering data. *2012 Sustainable Internet and ICT for Sustainability, SustainIT 2012*, (October), 2012.

[RESM17] Annie Ibrahim Rana, Giovani Estrada, Marc Sole, and Victor Muntes. Anomaly Detection Guidelines for Data Streams in Big Data. *Proceedings - 2016 3rd International Conference on Soft Computing and Machine Intelligence, ISCMI 2016*, pages 94–98, 2017.

[RSG+14] Matthias Reif, Faisal Shafait, Markus Goldstein, Thomas Breuel, and Andreas Dengel. Automatic classifier selection for non-experts. *Pattern Analysis and Applications*, 17(1):83–96, 2014.

[Sch19] Peter Schneider. Do's and Don'ts of Distributed Intrusion Detection for Industrial Network Topologies. *Proceedings - 2019 IEEE International Conference on Big Data, Big Data 2019*, pages 3222–3231, 2019.

[SFT+20] Jakapan Suaboot, Adil Fahad, Zahir Tari, John Grundy, Abdun Naser Mahmood, Abdulmohsen Almalawi, Albert Y. Zomaya, and Khalil Drira. A Taxonomy of Supervised Learning for IDSs in SCADA Environments. *ACM Computing Surveys*, 53(2), 2020.

[SG19] Peter Schneider and Alexander Giehl. *Realistic data generation for anomaly detection in industrial settings using simulations*, volume 11387 LNCS. Springer International Publishing, 2019.

[SS97] Hochreiter Sepp and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, (9):1735–1780, 1997.

[VCA+20] Ivan Vaccari, Giovanni Chiola, Maurizio Aiello, Maurizio Mongelli, and Enrico Cambiaso. Mqttset, a new dataset for machine learning techniques on mqtt. *Sensors (Switzerland)*, 20(22):1–17, 2020.

[WR] Christian Wressnegger and Konrad Rieck. Z OE : Content-based Anomaly Detection for Industrial Control Systems. pages 1–12.

[YZU+17] Chin-Chia Michael Yeh, Yan Zhu, Liudmila Ulanova, Nurjahan Begum, Yifei Ding, Hoang Anh Dau, Diego Furtado Silva, Abdullah Mueen, and Eamonn Keogh. Matrix profile i: All pairs similarity joins for time series: A unifying view that includes motifs, discords and shapelets. pages 1317–1322, 2017.

[ZRA20] Yue Zhao, Ryan A. Rossi, and Leman Akoglu. Automating Outlier Detection via Meta-Learning. 2020.

[ZRL96] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. BIRCH: An Efficient Data Clustering Method for Very Large Databases. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 25(2):103–114, 1996.

Appendix

1 Formulas

This section contains formulas that are named but not explicitly defined in the thesis text.

Definition .1. The **z-Normalization** of a single time series $x = x_1, ..., x_n$:

$$\hat{x}_i = \frac{x_i - \mu_x}{\sigma_x} \tag{1}$$

Where μ_x is the series' mean and σ_x is the series' standard deviation. □

Definition .2. The **mean** of a single time series:

$$\mu_x = \frac{\sum_{i=1}^{n} x_i}{n} \tag{2}$$

□

Definition .3. The **standard deviation** of a single time series:

$$\sigma_x^2 = \frac{\sum_{i=1}^{n} x_i^2}{n} - \mu_x^2 \tag{3}$$

□

Definition .4. The **mean squared error** between a observed value list x and a predicted value list $\hat{x}$:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(x_i - \hat{x}_i)^2 \tag{4}$$

□

2 Concepts

This section contains relevant concept definitions.

2.1 Rectified linear activation function

Definition .5. The rectified linear activation function can be defined as :

$$\text{rectifier}(x) = \max(0, x) \tag{5}$$

□

This function replaces all negative input with 0 and passes positive input without modification [GBB11].

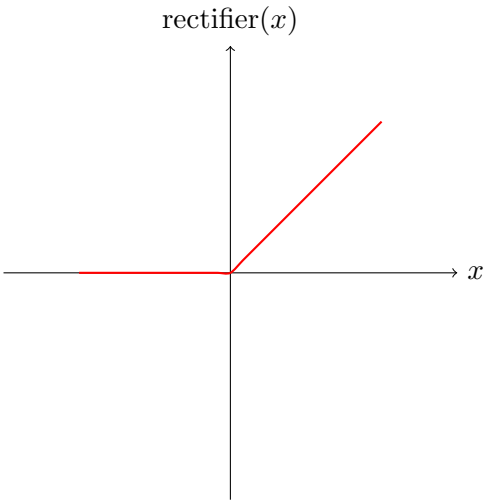

Figure 1: Rectified linear activation function.

2.2 Sigmoid activation function

Definition .6. The (logistic) sigmoid activation function can be defined as :

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{6}$$

□

This function models the biological activation of a neuron. It is a common activation function for neural networks [GBB11].

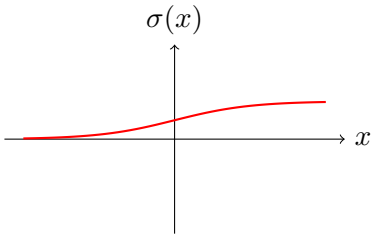

Figure 2: Logistic sigmoid function.

2.3 The tanh activation function

Definition .7. The tanh activation function can be defined as :

$$\tanh(x) = \frac{\sinh(x)}{\cosh(x)} = 1 - \frac{2}{e^{2x}+1} \tag{7}$$

□

This function also models the biological activation of a neuron similar to the sigmoid activation function [GBB11].

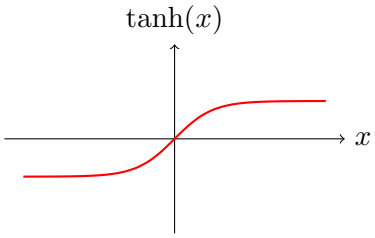

Figure 3: Tanh function.

3 Lists and Tables

In this section we provide lists and referenced tables containing additional information.

Hardware	Values
CPU	Intel Core i7-8565U
RAM	16GB

Table 1: Specifications of *Computer-1*.

Hardware	Values
CPU	AMD EPYC 7232P
RAM	16GB

Table 2: Specifications of *Computer-2*.

The following configuration of features is used to characterize data sources before clustering. Further documentation[5] is available online.

Listing 1: Clustering features

```
fc_parameters = {
        "variance_larger_than_standard_deviation": None,
        "ratio_beyond_r_sigma": [{"r": 2.5}],
        "large_standard_deviation": [{"r": 0.1}],
        "has_duplicate": None,
        "median": None,
        "mean": None,
        "standard_deviation": None,
        "variance": None
}
```

[5]https://tsfresh.readthedocs.io/en/latest/text/feature_extraction_settings.html

Sequence length	Precision	Recall	F1-Score	Accuracy
5	**0.6**	1.0	**0.643**	0.997
10	0.537	1.0	0.568	0.997
15	0.411	1.0	0.478	**0.998**
20	0.313	1.0	0.353	0.98
25	0.457	1.0	0.53	0.997
30	0.068	1.0	0.101	0.885
35	0.348	1.0	0.415	0.991
40	0.08	1.0	0.133	0.979
45	0.098	1.0	0.171	0.989

Table 3: Average LSTM detection scores depending on input sequence size with tanh activation function.

Sequence Length	Precision	Recall	F1-Score	Accuracy
5	**0.498**	1.0	**0.546**	0.992
10	0.481	1.0	0.523	0.993
15	0.108	1.0	0.158	0.939
20	0.287	1.0	0.332	0.974
25	0.336	1.0	0.403	**0.996**
30	0.077	0.75	0.124	0.865
35	0.081	0.75	0.139	0.99
40	0.055	0.75	0.096	0.963
45	0.031	0.725	0.058	0.933

Table 4: Average LSTM detection scores depending on input sequence size with relu activation function.

Sequence Length	Precision	Recall	F1-Score	Accuracy
5	**0.482**	1.0	**0.508**	0.98
10	0.314	1.0	0.359	0.98
15	0.123	1.0	0.176	0.942
20	0.32	1.0	0.366	0.976
25	0.128	1.0	0.193	0.976
30	0.081	1.0	0.12	0.883
35	0.113	1.0	0.184	0.916
40	0.087	1.0	0.144	0.97
45	0.125	1.0	0.165	0.938

Table 5: Average LSTM detection scores depending on input sequence size with sigmoid activation function.

Brute force in seconds	Representative in seconds	Clusters	Overlap	Datasets
353.25	260.82	1.0	1.0	2
432.92	269.08	1.0	1.0	3
502.83	346.03	2.0	1.0	4
611.39	361.72	2.0	1.0	5
709.03	376.12	2.0	1.0	6
764.80	434.18	3.0	1.0	7
803.02	522.87	4.0	1.0	8
890.58	431.19	3.0	1.0	9
995.74	421.50	3.0	1.0	10

Table 6: Comparison of the elapsed time of brute force and representative.

Windowsize	Kernel Density Estimation	Matrix Profile	LSTM
50	0.004	0.001	1.224
150	0.01	0.001	4.466
250	0.016	0.001	7.605
350	0.022	0.001	10.933
450	0.027	0.001	14.13
550	0.034	0.001	16.517
650	0.038	0.001	17.996

Table 7: Average time needed for feature computation, online clustering and anomaly detection method execution in relation to the size of the data window.

Method	Dataset ID	Dataset name	Size	F1-Score	Accuracy	True positives	False positives
KDE	1	ec2_cpu_utilization_5f5533	733	0.667	0.999	1	1
	2	ec2_cpu_utilization_24ae8d	733	0.222	0.99	1	7
	3	ec2_cpu_utilization_53ea38	733	0	0.996	0	2
	4	ec2_cpu_utilization_77c1ca	733	0	0.918	0	59
LSTM	1	ec2_cpu_utilization_5f5533	733	0.054	0.952	1	35
	2	ec2_cpu_utilization_24ae8d	733	0.154	0.985	1	11
	3	ec2_cpu_utilization_53ea38	733	0	0.996	0	2
	4	ec2_cpu_utilization_77c1ca	733	0	0.911	0	64
MP	1	ec2_cpu_utilization_5f5533	733	0.036	0.928	1	53
	2	ec2_cpu_utilization_24ae8d	733	0.014	0.802	1	145
	3	ec2_cpu_utilization_53ea38	733	0	0.959	0	29
	4	ec2_cpu_utilization_77c1ca	733	0.015	0.817	1	134

Table 8: Evaluation results of prototype live detection on four datasets.

Method	Dataset ID	Dataset name	F1-Score	Accuracy
KDE	1	ec2_cpu_utilization_5f5533	1.0	1.0
	2	ec2_cpu_utilization_24ae8d	1.0	1.0
	3	ec2_cpu_utilization_53ea38	1.0	1.0
	4	ec2_cpu_utilization_77c1ca	1.0	1.0
LSTM	1	ec2_cpu_utilization_5f5533	1.0	1.0
	2	ec2_cpu_utilization_24ae8d	0.8	0.9999
	3	ec2_cpu_utilization_53ea38	0.8	0.9996
	4	ec2_cpu_utilization_77c1ca	0.8	0.9997
MP	1	ec2_cpu_utilization_5f5533	0.0	0.998
	2	ec2_cpu_utilization_24ae8d	0.0	0.9994
	3	ec2_cpu_utilization_53ea38	0.0	0.998
	4	ec2_cpu_utilization_77c1ca	0.0	0.9987

Table 9: Average F1-Score and accuracy values of detection methods during Method Evaluation on different datasets.

4 Figures

This section contains figures that contain additional information that is not relevant to the main text but adds extra context.

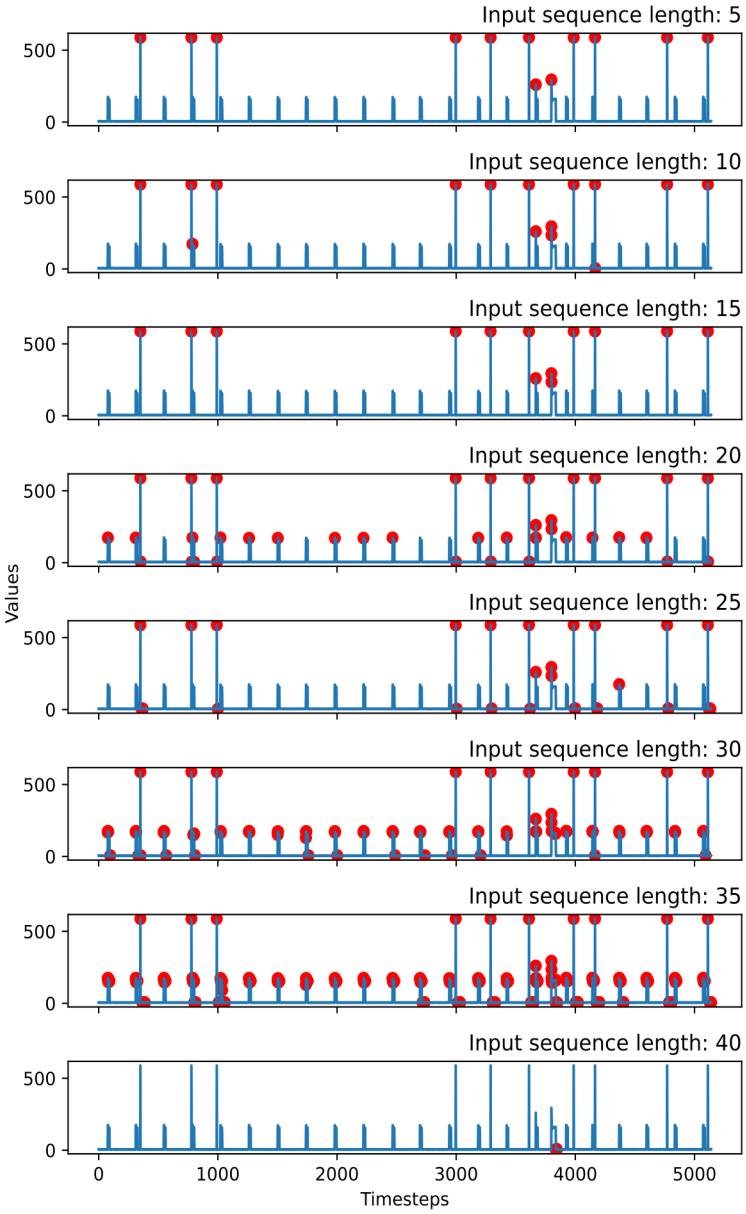

Figure 4: Different anomaly detection results are achieved by LSTM detection depending on the size of the input sequence. Values larger than 500 are injected anomalies. Red dots highlight values that are deemed anomalous by LSTM anomaly detection.