# Towards Generalizability and Comparability in Predictive Maintenance

Master Thesis

presented by
Philipp Steinrötter
Matriculation Number 1635016

supervised by
Cahit Bagdelen
SAP SE

submitted to the
Data and Web Science Group
Prof. Dr. Paulheim
University of Mannheim

August 2020

# Abstract

With a huge revival of artificial intelligence, companies' interest in leveraging Predictive Maintenance to reduce the downtimes of their industrial equipment is on the rise. Accordingly, a lot of research is currently underway. Most research, however, focuses on finding suitable methods for only a very narrow set of machine types, which limits the generalizability of the algorithms and significantly increases the cost of deployment in an operational environment. Furthermore, research is hampered by the lack of comparability of the algorithms due to the diversity of the methods used and the sensitivity of the required data.

Proposed in this thesis is both a generalizable and unsupervised algorithm for predicting machine failures and a benchmark for sharing datasets and comparing algorithms in the field of Predictive Maintenance. As the basis for both contributions, an unprecedented definition of Predictive Maintenance as a problem of artificial intelligence is given. The benchmark provides a framework within which researchers can easily contribute datasets and evaluation metrics and implement and compare their algorithms in an open but fair way. At its first publication, it already contains several open datasets and a novel cost-based evaluation metric specifically designed for Predictive Maintenance. The failure prediction algorithm is a multivariate Hierarchical Temporal Memory-based anomaly detection algorithm and uses algorithms derived from the Extreme Value theory to convert the anomaly likelihood into a discrete value indicating whether or not a maintenance alarm should be triggered.

Experimental results suggest the validity of both the benchmark including the cost-based evaluation metric and the unsupervised failure prediction algorithm. While the former represents a significant step towards comparability and open research in the field of Predictive Maintenance, the latter exhibits valuable characteristics such as generalizability, interpretability and adaptability.

# Contents

# List of Algorithms

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Traditionally, there have been three ways to organize maintenance: Corrective Maintenance, Preventive Maintenance and Predictive Maintenance (PdM). With corrective maintenance, sometimes also called run-to-fail [111] or breakdown maintenance [76] in literature, the maintenance operation is performed after the machine failed. The obvious downsides are safety concerns and high costs due to the resulting downtime. With preventive maintenance, the degrading parts are replaced based on a periodic timeframe, which leads to a non-optimal utilization of resources. With Internet of Things (IoT) technologies now slowly converging to its plateau of productivity, companies interest in leveraging Artificial Intelligence (AI) to reduce downtime and increase the productivity of their industrial equipment is on the rise. [100] PdM monitors the current status of a machine and tries to predict a failure, so that the maintenance operations can be planned and implemented in the most cost-effective manner. [61] A study from Roland Berger predicts the market value of PdM to be six billion dollars by 2022, being one of the first IoT solutions that are becoming de-facto standard in the industry. [44] Accordingly, researchers continue to work on new and improved methods to further reduce the complexity and cost of implementing PdM.

## 1.1 Predictive Maintenance by Means of Anomaly Detection

Knowing that a machine will fail can be based on two reasonings. First, based upon knowledge about how the machine has behaved in the past when a failure is imminent, recurring patterns can be learned by a supervised algorithm using historic failure data. When a known pattern is detected again, the machine will likely fail soon. Second, by knowing how the machine behaves normally, any deviation

from normal can be measured and detected using anomaly detection algorithms. The latter will be the focus of this thesis. Generally speaking, detecting anomalous events requires two stages. First, a definition of normality is derived from the most frequently occurring patterns, which then allows an assessment of the extent to which the currently measured data appear abnormal. In prediction-based anomaly detection on streaming data, these judgements are based upon predictions of what patterns are expected given the learned definition of normal. [13, 17] If the predictions are not accurate, the current state is given a high anomaly score and if the anomaly score exceeds a previously set threshold, the current state is considered abnormal. Within the context of PdM, an abnormal state is an indication that something is wrong with the machine and a maintenance activity is scheduled.

## 1.2 Research Motivation

Although a lot of research is emerging recently, practical implementations of PdM remain rare due to several open issues which are discussed in detail in section 3.1.3. Currently, research in PdM focuses more on improving the performance of algorithms on a specific machine type than on finding generalizable models that can be applied to a wide range of machine types, which discourages many companies from establishing PdM due to high initial effort of covering all machines. This is further amplified by the lack of publicly available data sets, forcing researchers either to work on the same few available data sets or to collaborate with companies, which limits publishability and thus reproducibility. Another challenge in PdM lies in the difficulty of obtaining failure data. Since a machine barely fails, long data acquisition phases are required for significant historic failure data to be gathered. Despite this, current research is focused on the use of supervised algorithms, thus limiting the applicability of the methods to later stages of a PdM implementation. Furthermore, since hardly any ground truth labels are available except for the time of failure, researchers either rely on experts or use custom procedures to obtain the labels required for the supervised algorithms. However, unsupervised algorithms are also associated with considerable challenges. First, because they use less information for the same task, they perform, at least in theory, poorer than supervised alternatives if historic failure data is available. Moreover, they output a relative anomaly score only, and a threshold is required to mark a condition as anomalous. In recent literature, this problem is mostly solved by setting a threshold based upon knowledge of the data. [71, 70, 39] But especially during the acquisition phase, when little or no data is available and therefore unsupervised algorithms should outperform supervised algorithms, such knowledge is difficult or impossible to obtain. In addition, a fixed threshold is not suitable for non-stationary data streams

where the context may change and evolve over time, such as when the production process in which the machine is installed changes.

In summary, research in PdM is hampered by a lack of comparability due to a wide variety of evaluation metrics and limited data availability. Moreover, since most algorithms are developed with a narrow focus on a specific machine type, often enforced by limited data availability, and with the requirement for labelled data, they lack generalizability. The work in this thesis attempts to provide contributions towards both generalizability and comparability in PdM.

## 1.3 Research Contributions

The main contributions of this work are twofold. First, a novel benchmark for the comparison of algorithms on PdM is designed, developed and published, which represents a significant step towards comparability. Second, a novel failure prediction algorithm is proposed, which promises to have several properties useful for practical implementations of PdM, including generalizability. To design the algorithm, aspects of the Hierarchical Temporal Memory (HTM) theory, introduced in section 2.3, and the Extreme Value Theory (EVT), introduced in section 2.4, are combined into a multivariate, unsupervised anomaly detection algorithm that outputs a discrete label indicating whether or not a maintenance alarm should be triggered without requiring historical data or knowledge of the machine. To summarize, the principal contributions of this thesis are:

1. A novel definition of PdM as an AI problem that introduces and describes important properties to be considered when working in the field.

2. In preparation for the benchmark, all publicly accessible datasets known to the author are collected, pre-processed and their data format unified.

3. A novel benchmark designed as a common platform for researchers to share and compare algorithms, evaluation metrics and datasets in an accessible and transparent manner.

4. As part of the benchmark, a novel window-based assessment metric that can be applied to any type of algorithm is proposed. The metric was designed with respect to the above-mentioned definition of the PdM as a AI problem and inspired by the economic perspective of PdM.

5. In preparation for the proposed multivariate anomaly detection algorithm, the performance of htm.core, a community-driven implementation of the HTM theory, has been improved by 200% for univariate anomaly detection

problems. As part of this improvement, a parameter optimisation framework for Numenta Anomaly Benchmark (NAB), a benchmark for univariate streaming anomaly detection, was introduced into the open-source repository.

6. Since related work use HTM-based algorithms mostly for the detection of univariate anomalies, an extension for multivariate HTM-based anomaly detection is proposed, implemented and discussed.

7. To convert the relative anomaly scores produced by the multivariate HTM-based algorithm into discrete labels indicating whether or not to raise an alarm, Streaming Peaks-Over-Threshold with Drift (DSPOT), an EVT-based automatic thresholding algorithm, is adapted for use in highly dynamic scenarios.

8. A novel multivariate, unsupervised anomaly detection algorithm that is capable of issuing discrete labels instead of just relative anomaly scores, thereby triggering alarms without historical failure data being available.

## 1.4 Outline

This thesis is organized into seven main chapters. Following this introduction, chapter 2 provides a literature review of the theory necessary to understand this thesis and the underlying concepts. The main subjects are PdM, anomaly detection in time series, HTM and EVT. In the context of PdM, the above-mentioned definition of PdM as an AI problem is presented.

Chapter 3 is an exploration of related work on PdM, anomaly detection and benchmarks. Particular attention is paid to further elaborating open issues in PdM, explaining current implementations of the HTM theory and introducing NAB as a source of inspiration for the benchmark proposed in this paper.

Chapter 4 provides a detailed explanation of the benchmark proposed in this paper. This includes an overview of the requirements and their motivations, the design and implementation of the framework and the content made available on initial publication. Special care is taken to explain and formulate the novel evaluation metric.

Chapter 5 presents the HTM-based multivariate anomaly detection algorithm for failure prediction. This includes discussions about its design, a description of the contributions made in the HTM community and the formulation of the respective algorithms.

Chapter 6 presents several experiments that were performed to thoroughly examine both the failure prediction algorithm and the benchmark. This includes a description of the respective experiments and a discussion of their results.

Chapter 7 is a summary of the results obtained in Chapter 6 and the conclusions that can be drawn from them. Finally, possible directions for further research are presented in detail.

# Chapter 2

# Background

## 2.1 On Predictive Maintenance

Generally speaking, PdM is a maintenance strategy based on machine and process data and is therefore classified as an Industry 4.0 technology. For a better understanding of the research area, the following section aims to provide a deeper understanding of PdM as a research topic by providing a definition, demarcations to related areas and the research history of PdM. Furthermore, the AI problem behind the maintenance strategy PdM is investigated and defined.

### 2.1.1 Definition and Delimitation

In this thesis, PdM will be interpreted as those activities involving continuous or periodic monitoring in order to forecast component degradations so that as needed, planned maintenance can be performed prior to equipment failure. [38, 75, 93] Here, monitoring means watching carefully to determine the current health status and to predict the future status using forecasting. Degradation describes the deficiencies relative to the best possible health status, to which the machine is returned to by applying maintenance. The execution of a planned maintenance activity indicates that a lead time is required for spare part management and scheduling activities, since, naturally, applying maintenance at the time of failure does not help. Performing said maintenance activity before equipment failure means that the maintenance must be carried out before a functional failure of the machine. Put simply, PdM tries to predict the failure of a machine to apply maintenance operations before the failure actually occurs.

A nearly equivalent yet different term is Prognostics and Health Management (PHM), which describes a discipline that pursuits to predict and prevent failures. In particular, prognostics cares about predicting the future condition of a product.

Health management is defined as the process of measuring and monitoring the degree of degradation to carry out informed system lifecycle management. [99, 82] The key differentiator compared to PdM is that PHM is considered to be a framework of methodologies applicable to any kind of system, including non-industrial ones, while PdM is focussed on industrial equipment only and incorporates knowledge of the future state of machinery into the maintenance processes. Hence, PdM could be coined "industrial prognostics". [7]

### 2.1.2 History

The definition given above already indicates various possibilities of how PdM can be performed. Figure 2.1 is a custom representation of these methods and categorizes them into three categories - visual inspections, real-time condition monitoring, and AI.

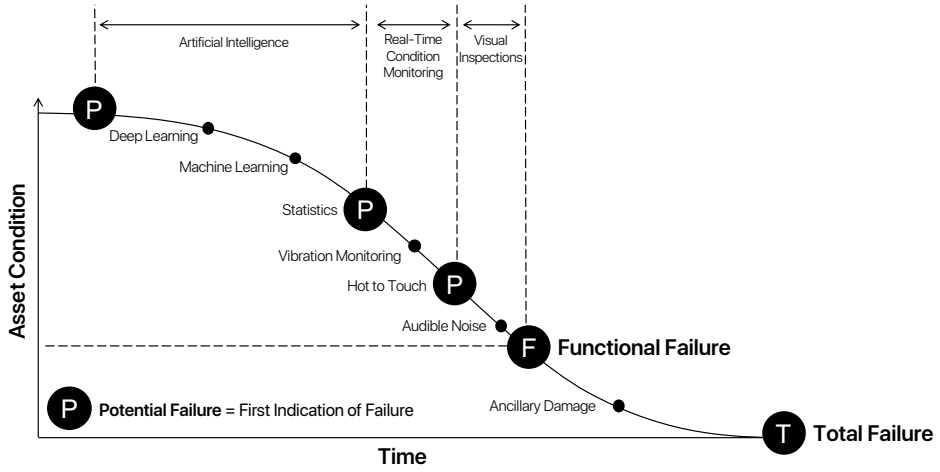



Figure 2.1: Predictive Maintenance methods and their effectiveness

The first category of PdM is solely human-driven. Inspectors periodically check the machine using instruments as well as their senses and, in combination with their personal experience, determine whether it is going to sustain a functional failure soon. First indications could be an audible noise due to the degradation of a bearing within rotating equipment. The data is usually recorded manually in asset management software solutions, e.g. through measurement and counter readings. [84] It goes without saying that the required data are univariate data and their collection manual. Furthermore, the time to react with a proper maintenance action to prevent a functional failure is short and does not allow a dynamic planning of

maintenance events.

The second level, which is also referred to as model-based PdM, was enabled by Cyber-Physical System (CPS), which refers to "a new generation of systems with integrated computational and physical capabilities" [22] and describes the integration of computational capabilities into the physical world e.g. by leveraging sensor data. [74] Machine health is either monitored through simple rules on data streams, e.g. by setting a threshold on the vibration, or by statistical methods such as stochastic filtering or Gamma processes on univariate time series data to model the degradation process. [76] Although these methods do not require a lot of historical data, their applicability is limited, as especially large and complex machines cannot be modelled accurately. [89] Furthermore, they are bound to expert knowledge and do not scale well.

The third category is AI methods, which no more rely on the experience of the reliability engineer and statistical modelling and rather learn models through data from sensors recording the relevant features. Consequently, more sophisticated Machine Learning (ML) and Deep Learning (DL) methods are preferred nowadays. For most applications, AI methods have shown to allow for the most time to dynamically plan the required maintenance events in order to prevent a failure and are therefore preferred. However, if the degradation process can be modelled accurately, model-based methods still succeed. [63]

### 2.1.3 Predictive Maintenance as an Artificial Intelligence Problem

From the AI perspective, PdM has so far only been loosely defined as a problem of failure prediction in the context of maintenance. Moreover, at least to the humble knowledge of the author, literature currently lacks a clear definition of PdM as an AI problem. Instead of considering PdM as a generalizable problem, most of the literature focuses on the solution of the problem given by a specific use case, leading to a narrower approach limited to the dataset. Furthermore, recent studies on PdM are based on publicly available datasets recorded in laboratories. [111, 74, 32] Therefore, they naturally lack generalizability and reach their limits in an operating environment where the environmental condition, recording frequency, and data quality varies. [82] Consequently, a generalizable definition of the AI problem behind PdM is required and will be laid out hereinafter, along with its properties.

#### Formal Definition

Formally, PdM attempts to predict a functional failure that occurs at time $t_F$ in order to plan appropriate maintenance work and prevent the failure. The time be-

fore the failure that is required for spare part management and scheduling activities is defined as the lead time $L$, which is a model parameter that depends on business requirements. Hence, it should be set individually to the time delta it takes to schedule and perform maintenance activities for the equipment at hand. Any prediction after $t - L$ is too late and will not prevent the failure. Consequently, to ensure minimal cost, the failure should be predicted just before $t_F - L$.

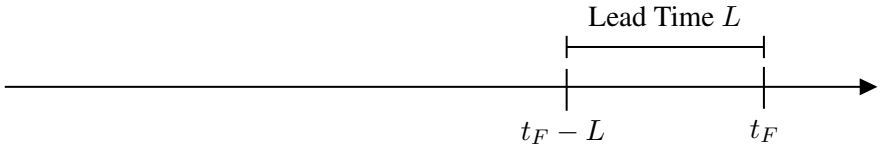



Figure 2.2: Illustration of the lead time

### Properties

**Provision of Ground Truth Difficult** Typically, predictive models are learned from a ground-truth training set that contains a large set of examples, each consisting of an instance and the respective label. In PdM however, the relationship between patterns in the data preceding an equipment failure and the failure itself naturally is unknown, meaning that neither the occurrence, nor the time or the relevancy of an anomaly regarding a failure is known. Hence, it is unknown during training whether some patterns that are detected before a failure occurs are related to the failure or just random, meaning that there is an unknown time lag between any anomalous time window and the succeeding failure. Furthermore, some failures occur due to a continuous degradation of its components, others due to a sudden event such as an unexpected external impact. [21, 85] The sheer variety of patterns is exemplified in Figure 2.3. Each run-to-failure expresses another type of pattern, ranging from a very predictable, continuous to a very sudden degradation. Consequently, besides the time of the failure itself, no information is available that can be used for labelling the instances. This is approached in various ways in literature. For regression tasks, such as predicting the Remaining Useful Live (RUL), every member $X_t$ of a time series $X = \{X_t : t \in T\}$ is labelled with $\max(T) - t$. [32, 63] However, if treating PdM as a regression problem, the accuracy of the predicted RUL strictly depends the linear distance between the current time $t$ and the time of machine failure, $\max(T)$, and therefore on the weak assumption of linear degradation. Consequently, using the predicted RUL at early stages in the equipment lifetime can lead to wrong decision making and high costs due to a low prediction accuracy. [74] For classification tasks, some research solves this problem by manually labelling the data based on expert knowledge. However, such a

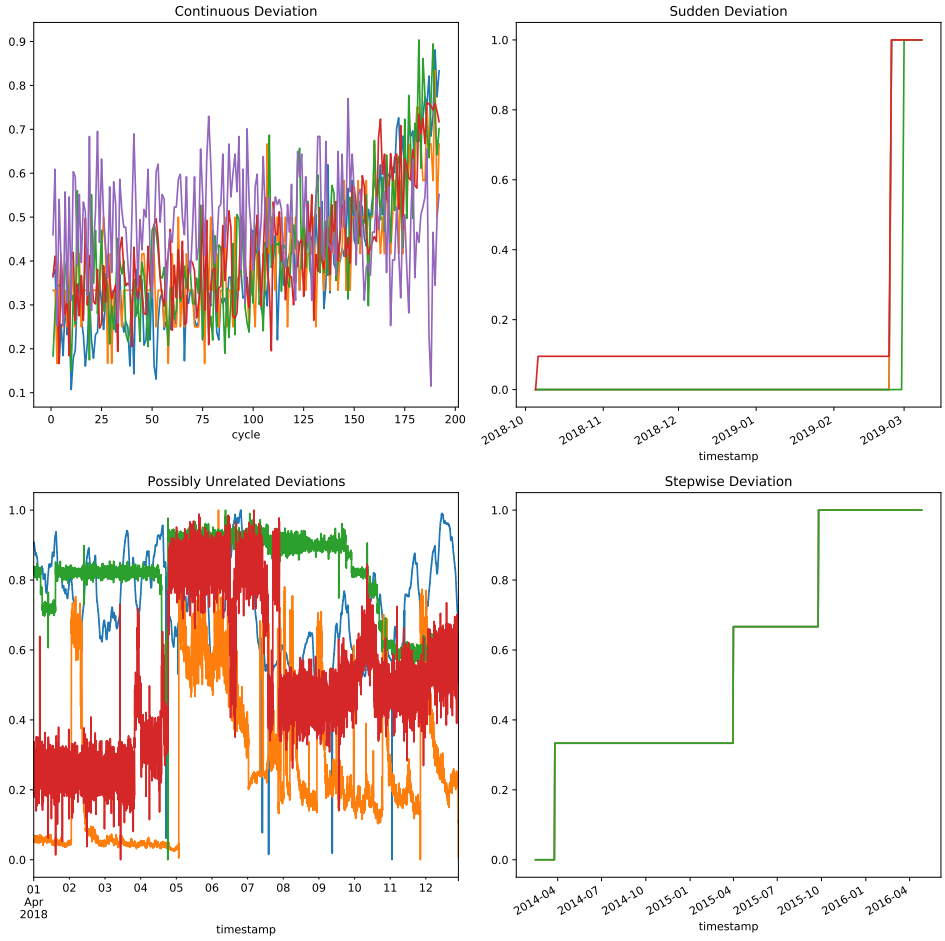



Figure 2.3: Four run-to-failures of different machines of different types, showcasing the range of properties that the instance-to-label relationship can take. Each subplot shows time-series data of a complete run-to-failure, with the $x$-axis extending from equipment start up to equipment failure. Note that for reasons of clarity, only some of the available dimensions are shown in each plot. [21, 85]

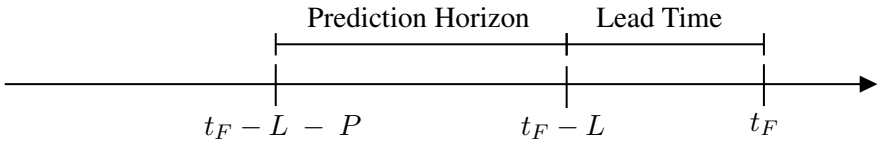



Figure 2.4: Prediction horizon and lead time

procedure is very costly, does not scale well, and still does not guarantee correct labelling. [82] Another approach is introducing a prediction horizon, sometimes also called prediction window or prediction period. The prediction horizon defines how far into the future failure can occur and the prediction is still be considered correct. Therefore, any instance within the prediction horizon is considered a positive example. Naturally, such a procedure is based upon the assumption that any pattern observable before a failure occurs is relevant. In most literature, a prediction horizon of size $P$ is defined so that any alarm raised within $[t_F - P; t_F]$ is considered correct. [74, 110, 67] However, taking the business context of PdM into account, a lead time $L$ is required for scheduling activities. Hence, in this work, the prediction horizon shall be defined as the time $P$ before the lead time $L$, in which a maintenance activity would repair the machine and prevent its failure. Figure 2.4 illustrated the prediction horizon and the lead time. Any alarm raised before $t_F - L - P$ would lead to unnecessary maintenance activities and costs.

**Multiple-Instance-Single-Label Learning** In standard classification tasks, each instance is categorized with a single label. In PdM however, only a single positive prediction needs to occur within the prediction horizon for the failure to be predicted, while multiple positive predictions do not change the positive outcome. To put it into a maintenance context, an alarm must be raised at least once within the prediction horizon to trigger a maintenance repair activity which prevents the failure. Any alarm raised after the first within the prediction horizon will be ignored by the reliability engineer as the related maintenance is already being executed. Hence, raising further alarms does not change the positive outcome. Consequently, based on the taxonomy of non-standard classification problems provided in [53], PdM can be considered as a Multiple-Instance-Single-Label (MISL) learning problem. The idea of defining PdM as a MISL problem was first expressed by Cahit Bagdelen.

**Mixed Type Input** In most practical implementations of PdM, data from various heterogeneous sources are merged in a streaming manner. For instance, the current physical condition such as the temperature or the pressure of different parts of

the machine is frequently recorded by sensor systems. Additionally, information about the current configuration, the operating status as well as process information is extracted from Supervisory Control and Data Acquisition (SCADA) systems. For example, a hot rolling process would yield information about the type of steel that is currently being processed and the respective configuration that is applied to the individual drums. Furthermore, business-related data is gathered from Enterprise Resource Planning (ERP) systems, including spare part availability and other logistics-related information. [67, 74, 82, 97] Consequently, the data typically consists of multivariate time-series data from sensors, event data in form of categorical values as well as other, scalar and non-scalar time-series data such as switches, counters and changing configuration settings. For a successful PdM implementation, the data has to be harmonized, certainly adding another level of complexity to it. The process of harmonization is usually referred to as data-to-information bride, feature engineering, or feature extraction. [82, 113, 59]

**Collective Nature of Anomalies** Another property of PdM is the collective nature of anomalies. Due to the high dimensional data input, each time series has to be handled not only individually but also collectively, since minor problems in one part of a system can easily lead to a functional failure if they cascade into other areas. Furthermore, because a machine usually consists of multiple interacting components that form a unified whole, many of the individual time-series measured at different components correlate with each other, adding another level of complexity to PdM. [17, 98]

**Dynamic Assumptions** Usually, a machine is operating within a larger system such as a production line, which undergoes frequent changes. As an example, a robot arm in a car factory may be required to lift the heavy door of a truck instead of a car, resulting in vastly different yet healthy sensor readings. Consequently, another important property of PdM is the dynamic environment, in which any algorithm must learn and adapt properly. Any practical implementation must involve a clear understanding of the process or product from which the data is collected, the collection process as well as how the measurements are physically obtained. Data recorded in a protected laboratory setting mostly misses this important property of PdM. Hence, laboratory-fitted models have operational limits. [111, 82]

Summarizing the above-mentioned characteristics, PdM can be considered a complex problem, located at the junction of AI and business with a multitude of points of contact in both directions. For a PdM implementation to be successful, all properties have to be taken into account.

## 2.2 On Time-Series Anomaly Detection

In general, an anomaly can be defined as any pattern that deviates significantly from an established and recognizable norm to distinguish it from the whole. Across different fields, anomalies are also known as outliers, inconsistent observations, exceptions, deviations, surprises, particularities or impurities. However, outliers and anomalies are the most commonly used terms of all domain-specific argot. [29] The first research aimed at detecting anomalous patterns in data can be dated back to 1969. At that time, anomaly detection was mainly used for data cleansing, since pattern recognition algorithms were quite outlier-sensitive. [45] After more robust algorithms had been developed, the research interest plummeted. However, around the year 2000, researchers became interested in the anomalies itself, as the respective signal usually contains interesting and actionable insights. Since then, the detection of anomalies has become an intensively investigated problem across various domains including network intrusion detection, health, and industrial damage detection. [29]

Note that the definition presented above makes no difference as to the nature of the anomaly. An anomaly that occurs due to a faulty sensor is just as interesting to detect as an anomaly that occurs due to a machine error.

Anomaly detection is approached in both, an unsupervised and a supervised manner. The unsupervised approach typically relies on prediction, clustering, or statistical modelling. The supervised approach typically uses binary classification to learn to distinguish between anomalous and non-anomalous data points. [49]

Anomaly detection on time series, sometimes also known as sequential anomaly detection, adds the temporal domain to the problem and is increasingly relevant in any field where data is recorded in the form of a time series and with relatively equidistant collection intervals, such as PdM. [30] However, especially in comparison to static anomaly detection, time series anomaly detection requires more sophisticated methods to capture and comprise the time-dependency of the data.

The following sections will provide a high-level overview of possible classes in which time series anomaly detection methods could be categorized. Furthermore, the different types of anomalies will be laid out, alongside with the strategies required to detect them. Since the data in PdM is multivariate, the challenges of detecting anomalies in multivariate data as opposed to univariate data are briefly discussed. Additionally, concept drift will be introduced as an important property of time series data, especially when recorded in a changing environment, as is the case with PdM, as derived in section 2.1.3.

### 2.2.1 Types of Anomalies

Hereinafter, three main types of anomalies will be introduced: spatial, contextual, and collective anomalies. This distinction has been made because different types of detection strategies are often required to reliably detect spatial, contextual, and collective anomalies. This is due to their different visibility characteristics, which are described below.

#### Spatial Anomalies

Spatial anomalies, sometimes also referred to as point anomalies, are the simplest type of anomaly. A data point is a spatial anomaly if it can be considered anomalous with respect to the rest of the data. These are the simplest type of anomalies, and often simple statistical measures and models are sufficient to detect them reliably. [29] Note that spatial anomalies do not have to be part of a time series, although they can be.

For example, looking at this unordered collection of datapoints $\{10.5, 12.4, 8.3, 23.9, 9.5\}$, one can easily state that $23.9$ is significantly unlike the rest of the data points and thus can be considered an anomaly without having any knowledge about how, in what order and in which context the data was recorded. Note that shuffling the data would not matter. Thus, in time series, a spatial anomaly can be defined as a local, short-term deviation from what is considered normal. [60]

In the context of PdM, an example for a spatial anomaly could be am unanticipated, sharp increase of the bearing temperature of a centrifugal pump to $150°$, after a baseline of $110°$ has been well established. In this case, the spatial anomaly might signify a broken casing and should trigger an alarm, followed by a corrective maintenance activity.

Note that not all spatial deviations from the norm are anomalies. Each data collection method does naturally have an irregular, mostly normally distributed, random noise, sometimes referred to as white noise, associated with it. [24] This is especially true in the context of PdM, where sensors usually record data in an environment with high stress. As an example, a vibration sensor on any machine on a factory floor is influenced by the vibrations generated by its surroundings. Therefore, any anomaly detection system must account for this noise to not label it as anomalous. Spatial deviations are only to be considered anomalous when the magnitude of the deviation is not normal with respect to the white noise. [60]

#### Contextual Anomalies

A contextual anomaly is a data point that may be expected in one context but is anomalous in another. For example, a temperature near zero is perfectly normal

during winter, but anomalous in summer. Naturally, the occurrence of a contextual anomaly requires a notion of context to be introduced by the structure in the dataset. As an example, in spatial data sets, the longitude and latitude information puts the data points into a geographical relation to each other. In time series, the context is induced by time and arranges the data points in a sequence, defining the time distance between them and the position of each data point within the sequence. Additionally, the context can be enriched by other attributes such as the type of material that is currently being processed in a machine. [29, 49] As an example, some sensor readings may be anomalous only in the context of a high-pressure operation.

Naturally, the difficulty in detecting contextual anomalies lies in their insignificance when viewed in isolation. Consequently, an anomaly detection algorithm has to have some kind of memory to be able to reasonably evaluate a data point with respect to its context.

### Collective Anomalies

A collection of data points that is anomalous only if considered together is termed a collective anomaly. An individual data point of that collection is not necessarily anomalous on its own. [29] As an example, a collective anomaly can be formed by medium-sized spatially deviating data points, which in themselves can rightly be treated as white noise, but are of explicit significance when they are directly adjacent. Note that in multivariate time series collective anomalies can also occur across dimensions. For example, a machine that emits abnormally high vibrations can be considered normal if it is running at high capacity, but a simultaneous rise in temperature can signal a faulty cooling system and should trigger an alarm.

The difficulty in detecting collective anomalies lies in the natural variety of occurring patterns since a collective anomaly is only detectable when both, spatial and contextual errors are taken into account. Hence, an algorithm must have some kind of error accumulation procedure to capture and evaluate the required information. [49] In addition, in multivariate time series, a collective anomaly can be formed by several patterns that occur close together but with a random time delay. For example, in a machine consisting of multiple components interacting with each other, problems may cascade to other areas. Hence, an anomaly detection algorithm must comprehend with random temporal delays between anomalies across dimensions. [17]

### 2.2.2 Univariate vs. Multivariate

Although it has already been declared that the data in PdM is multivariate, a brief distinction between univariate and multivariate anomaly detection will be given hereinafter to better understand the nature of the latter.

Univariate time series anomaly detection aims to detect unusual data points in a single dimension. As an example, in network intrusion detection, a high number of SYN packets can be an indication of an ongoing attack. Of course, a person looking at a plot of the data stream can also detect such an anomaly. Consequently, anomaly detection is mainly useful for automated monitoring and increasing human productivity. With multivariate data streams, finding an anomaly is much more difficult and requires more sophisticated methods or careful examination. Hence, multivariate time series anomaly detection, especially in data streams, is only possible with the help of algorithms, since a human is not able to grasp more than three dimensions efficiently enough.

### 2.2.3 Concept Drift

Concept drift is a phenomenon caused by changing data distributions due to dynamic and non-stationary environments. It is particularly important when analysing data streams since a change in the underlying distribution renders any model useless unless it adapts accordingly. Especially multivariate analysis is affected by the phenomenon, since the correlations between the individual data streams may also change over time. It can be distinguished between two types of concept drift. Real concept drift refers to a change in the conditional distribution of the output while the input remains unchanged. A prominent example is a change in news consumption when consumer interest changes. While the distribution of the incoming news articles mostly remains the same, the distribution of the interesting articles varies. On the contrary, virtual concept drift is defined as a change in the input, while the distribution of the output remains the same. Hence, while virtual concept drift is directly measurable from the input data, real concept drift can only be detected by observing the target data. [47, 78]

As discussed in section 2.1.3, a machine typically operates in a dynamic environment such as a production line. Any changes made to the production line can result in a change in the distribution of the recorded data without a change in the distribution of the target since the machine remains healthy. Consequently, virtual concept drift can be regarded as an important phenomenon when working with time series in the PdM domain.

In recent literature, concept drift is usually approached either by retraining and selection of static models based upon concept drift detection or by using an online

algorithm that is able to adapt by design. [83, 17] The detection of concept drift can be achieved for example by statistical process control methods like the Drift Detection Method presented in [46], or by model-based methods like proposed in [109] and [104]. In essence, these methods share the basic idea of comparing a preliminary model with the current status in order to assess the discrepancy. If this discrepancy is larger than a predefined threshold, a concept drift is detected and a new model should be trained. Another way to handle concept drift is to use an algorithm that can adapt to changing data streams by design. An example of such an algorithm is HTM, which is introduced in chapter 2.3 and serves as the basis for the algorithm that is proposed in chapter 5.

For a comprehensive overview of concept drift, its applications and detection techniques, the interested reader is referred to [58, 114, 47].

### 2.2.4 Classes of Time-Series Anomaly Detection

Several methods have been used to solve the problem of anomaly detection on time series data, including but not limited to statistical measures [102], rule-based systems [40], clustering-based methods [87], ML algorithms [64], Bayesian models [90], deep learning [96] and HTM [17]. Different real-world applications such as network intrusion detection or PdM have different underlying data distributions and requirements, imposing different demands on the respective method. For some real-world applications, some of these requirements are unrealistic and make the respective method unsuitable. An example of such requirements is the need for labels in the training data or a look ahead to evaluate past data as anomalies.

As discussed in section 2.1.3, in PdM, the provision of labels is difficult and a look ahead is required, as the objective is prevention rather than detection. Moreover, concept drift renders static models trained on past data obsolete, and labelling of incoming data is impractical, if not impossible.

Hereinafter, three mutually exclusive classes of time series anomaly detection are defined: unsupervised batch learning, online learning and hybrid learning. The classification is based on the above-mentioned requirements that an algorithm imposes on the data input and thus determines for which real-world applications the respective method can be used.

#### Unsupervised Batch Learning

The first class of anomaly detection algorithms is unsupervised batch learning. As the name indicates, any algorithm that falls into this category processes the data in batches and operates completely offline. Here, offline means that the processing takes place on already available and static data sets instead of on a constantly

evolving data stream. To detect the anomalies, the methods in this category implicitly assume that normal instances are far more common than abnormal ones. [20] Naturally, any unsupervised batch learner requires the entire dataset in advance, which is an unrealistic requirement for real-time scenarios such as network intrusion detection or PdM. Especially in a scenario where it is vital to detect an anomaly as early as possible it is not practical to discover anomalies only after a significant time has passed. Lastly, in times of big data, storing all or at least a sufficient amount of historic data is very costly and also impractical.

Nevertheless, batch learning algorithms do work well in any application where time is uncritical such as the retrospective analysis of historical data. Such an analysis is oftentimes used in analytical business applications such as data warehousing. For example, the analysis of an unusually high number of sales is useful to understand correlations in retrospect.

### Online Learning

In contrary to batch learning algorithms, online learning algorithms work directly on the incoming data stream. Hence, they evaluate on and optionally adapt themselves to every incoming datapoint in near real-time. The following characteristics presented in [17] resemble an ideal real-world anomaly detection algorithm:

1. Predictions must be made online; i.e., the algorithm must identify state $x_t$ as normal or anomalous before receiving the sub-sequent $x_{t+1}$.
2. The algorithm must learn continuously without a requirement to store the entire stream.
3. The algorithm must run in an unsupervised, automated fashion—i.e., without data labels or manual parameter tweaking.
4. Algorithms must adapt to dynamic environments and concept drift, as the underlying statistics of the data stream is often non-stationary.
5. Algorithms should make anomaly detections as early as possible.
6. Algorithms should minimize false positives and false negatives (this is true for batch learning algorithms as well).

Any algorithm that exhibits the above features can be classified as an online learner. Note that these points describe an ideal online algorithm. For example, some use cases may not require adaption to dynamic environments and concept drift.

### Hybrid Learning

Hybrid learning describes a class of algorithms that require an initial phase of offline learning on historical data. In essence, an algorithm of this class learns offline just like a batch learning algorithm but then uses the resulting model for evaluating the data stream in real-time.

There are three types of hybrid learning algorithms, and their difference lies in the way the model is trained. Semi-supervised hybrid learners assume that there is no anomaly within the training data, so all cases are considered normal. Thus, in the context of PdM, the model learns how the machine normally operates to detect a deviation from normal. Unsupervised hybrid learners function similarly to semi-supervised learners, except that the assumption that there are no anomalies is relaxed so that the training data may contain a few anomalies. Finally, supervised hybrid learners are classification-based algorithms that require a training set where all instances are labelled as either normal or abnormal.[29]

For all three types, after the model is trained, the key difference to an online learning algorithm is the lack of self-adaption since the model is static until another retraining is triggered. Consequently, the algorithm has to repeat the learning phase continuously to be able to handle concept drift. Hence, hybrid learning also requires the storage of historical data, although usually not as much as batch learning. For example, in the aforementioned example of a robotic arm in a car factory, when assigned to lift the heavy door of a truck instead of a car, the system first has to gather data again before the model can be retrained and deployed on the incoming data stream. Until then, the previously trained model will probably consistently raise alarms, since the distribution of the incoming data changes significantly while the model remains the same. Note that it is not always clear when to execute retraining, especially in applications where real concept drift occurs frequently. Most supervised anomaly detection algorithms can be classified as hybrid learners because an initial learning phase is required to train the model on labelled data before it can be deployed on the incoming data stream.

## 2.3 On Hierarchical Temporal Memory

HTM is a framework inspired by neuroscience that aims to develop intelligent models based on the human brain. It was initiated by Jeff Hawkins in [51] and has since experienced a continuously increasing research interest. The main difference to other AI approaches, such as neural networks, is the strong precept of deriving the entire design of the algorithm from the biological model, the human brain instead of just perceiving it as a archetype. Since such a venture is naturally very complex, the short term goal is to understand and implement how the neocortex is capable

of time series sequence learning and prediction. The work in this thesis uses the latest research on time series prediction with HTM for anomaly detection. Consequently, an introduction to HTM is given hereinafter, before it is discussed why HTM is preferred over neural networks, especially in the context of PdM. Since this thesis does not improve the algorithm itself but rather extends it, the following will serve as an introduction to HTM only. The interested reader is referred to [51] and [3] for more details.

### 2.3.1 A New Theory of Intelligence

The definition of intelligence is a well-discussed topic in literature, especially in the context of AI. While there is no generally accepted definition, leading textbooks refer to AI as the study of computational agents that act intelligently. An intelligent agent is any computer-based device that is able to perceive its current environment and take action to achieve a goal. [79] A similar definition was originally described in the Turing Test, which evaluates intelligence according to the extent to which the behaviour of a machine is not distinguishable from a human. [68] Consequently, it does not matter how a machine produces intelligent behaviour. As long as it can perform that behaviour, it is considered intelligent. In the Turing Test for example, how the machine tricks the human interrogator into thinking it is a human is irrelevant, as long as it is able to do so. Achieving a desired behaviour by any means has arguably shaped AI research significantly since its early beginnings.

However, HTM rejects the idea of behaviour-based intelligence by motivating first that behaviour is just the manifestation of intelligence, but not its primary definition. [51] For example, a person who lies in the dark and contemplates problems only in his head, without any physical behaviour, is considered intelligent. Numerous critically acclaimed thought experiments serve as an objection of behaviour-based intelligence, such as the Jukebox experiment and the Chinese Room experiment. The latter experiment, for example, argues that only because something exhibits the desired behaviour does not mean that it actually understands what it does. An interested reader is referred to [68] for more details about the experiments.

The theory of intelligence on which HTM is based argues that instead of behaviour, prediction is the essence of intelligence, meaning that only the ability to predict the future is a display of true intelligence. Hawkins illustrates in [51] how humans predict for virtually every task, e.g. by anticipating the trajectory of a flying ball when trying to catch it, how long it is going to take to get ready to go out, and how to answer a question given to them. Predictions are being made constantly on a less tangible scale, too. For example, humans predict how a pizza is going to taste when smelling it baking in the oven and, oftentimes even unconsciously, form

some kind of expectation. Hawkins argues that the consistency and accuracy of their predictions is how intelligent beings achieve a feeling of normalcy and make sense of everything that is around and within them. If a prediction is broken, a feeling of novelty arises. The ubiquity of predictions is the key to understanding its connection to intelligence. For example, the omnipresence of predictions also applies to seemingly trivial tasks like talking, walking on two feet or lifting a stone, for which the human brain has been trained over millions of years.

### 2.3.2 Biological Background

As mentioned above, HTM uses the human brain as the ultimate source of inspiration, while focussing on the neocortex and its time-series sequence learning capabilities first. Hereinafter, a brief introduction to the brain from a biological point of view will be provided, to then deduce the motivation to focus on the neocortex and explain its structure and function.

The human brain consists of multiple interconnected regions, each having their functions and responsibilities. The Cerebellum for example, located at the rear base of the brain, is responsible for fine-grained motor activities, such as pressing a piano key slightly more softly. Another important region is the amygdala, the emotional centre of the brain. It triggers emotional responses and attaches emotional significance to memory, making it difficult to forget. [8] The region responsible for everything we consider intelligent behaviour, such as perception, language, art and mathematics, is the neocortex. While all brain regions are critical to being human, Hawkins believes that an understanding of the neocortex is sufficient to build intelligent machines. He argues that in AI it is not of interest to build humans, but to understand intelligence and to build intelligent machines. Therefore, even if highly connected to the neocortex, other regions responsible for urges, hunger, muscles and emotions are not of primary interest for HTM. [51]

The neocortex itself consists of six layers that are visually distinguishable by a varying density and shape of the neurons. Each layer does have its own functionality and interaction with other layers. Furthermore, there is a columnar architecture perpendicular to the layers. This was first discovered by Mountcastle in [69], who coined these structures cortical columns. Of course, the columns do not have clear boundaries. Instead, their existence is derived from the observation that neurons organized in the same column are strongly connected and become active for the same stimulus. Every signal that reaches any layer within a column spreads up and down within the column across its layers. Figure 2.5 illustrates the appearance of the columns.

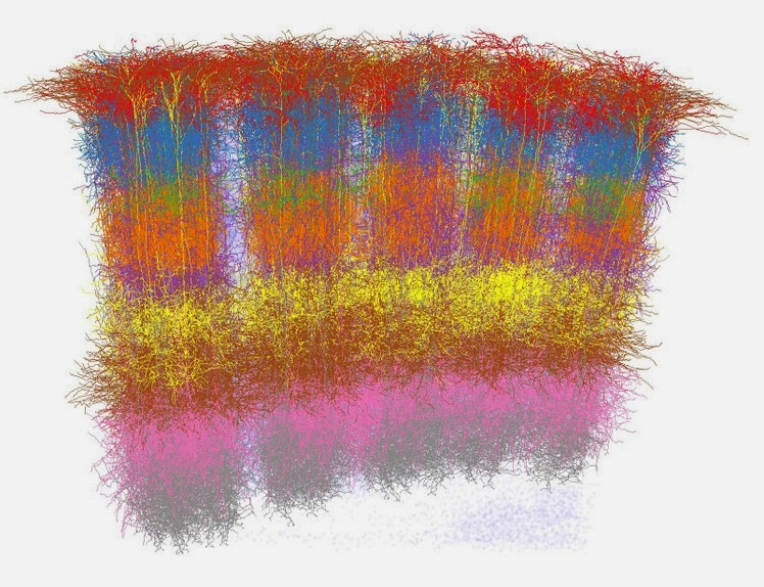


Figure 2.5: Cell-type-specific 3D reconstruction of five neighbouring columns in a neocortex region [23]

### 2.3.3 Core Principles

In the following some of the most important core principles of the HTM are outlined. These principles drive both theoretical research and implementation efforts within the HTM community. Note that this is only an overview that is intended to provide a basic understanding of the algorithms used in this thesis. For an exhaustive list, the interested reader is referred to [50].

#### Common Cortical Algorithm

As mentioned above, the neocortex consists of interconnected cortical columns. It is important to emphasize that it consists of these columns only, meaning that all regions of the neocortex are remarkably uniform in appearance and structure. This observation led Mountcastle to the conclusion that since the cortical columns are all the same, they must perform the same basic operation. The subtle difference between the regions is only predicated upon what they are connected to, not what their function is. [69] Consequently, all processing within the neocortex is done using cortical columns as a common unit of computation. Furthermore, since it is a common unit of computation, there must be a common underlying algorithm that is completely agnostic to the type of input. Whether the signal comes from

sound, vision or sensations, the algorithm that processes the information must be the same. From an evolutionary perspective, such a common design has two major benefits. First, since all input is processed the same way, the system is very flexible by nature. A remarkable experiment that illustrates this has been conducted by Paul Back y Rita. He has shown that when a person becomes blind and loses visual input, the brain is able to rewire the processing of the visual to sensory input. He developed a device that translates visual input recorded by a camera into tiny pressure points on the tongue. After a short period of time, the brain rewired the input and allowed the participant to "see" again. [51] Another benefit lays within the development of a new functional area of the brain. Evolution can simply copy the core design, as opposed to having to create a completely new design for every new purpose. [60]

### Sparse Distributed Representations

The human brain is capable of processing a quantity of information that is not comparable to any other existing processing device. Supported by various observations, cortical regions seem to use sparse activity patterns as a very flexible way of presenting information. In HTM theory, this representation is called Sparse Distributed Representation (SDR). The idea of SDR is derived from two remarkable observations. Firstly, the activity of neurons within the neocortex is sparse, which means that only a fraction of the neurons are active at the same time, while all others remain inactive. Secondly, the information is represented by a whole population of neurons. It is not an explicit representation of information such as in computer memory. Instead, any knowledge and sensory input is represented as a globally distributed pattern of sparse activity of neurons. [15] Based on the aforementioned properties, the HTM theory proposes SDR to be the common language of the brain, used to encode and process all incoming signals.

Current research focusses on binary SDRs, where each bit in a large bit vector represents a neuron. Active neurons are represented by $1$'s and vice-versa. Derived from observations, the sparsity level usually is around $2\%$, meaning that only $2\%$ of all neurons are active at any given time. [60] While each active bit does have a meaning, such as the presence of an edge within the current visual field, a human interpretable meaning can only be captured if the entire vector is considered. [15] For example, a face can only be detected if at least a significant amount of edges are visible and the respective bits active. Additionally, no single bit is critical, since the active bits are sparsely distributed over the entire vector. Hence, the face can still be detected, even if parts of it are hidden behind an obstacle. Furthermore, if two inputs activate the same bits, they are in a semantic relationship to each other. The higher the overlap between two SDR, the higher the semantic similarity between

the respective inputs.

From a mathematical perspective, such a representation does bring two major benefits with it. Firstly, if the size and sparsity is large enough, SDRs are very robust to noise, because flipping a few bits will not change the captured semantic meaning significantly. Similar properties can also be observed in real life, for example when playing the piano. The absence of one or the other note is not noticed by the audience, since there is a linear relationship between the overlapping of the notes played by the pianist compared to those of which the song is originally composed and the similarity between the two songs. The same linear relationship exists between the bit overlap of two SDRs and their semantic similarity. Secondly, although SDRs are very robust to noise, they do not suffer from accidental activation of neurons. If sparsity is maintained and the SDRs are of reasonable size, the chance of a false activation is very small due to a large number of possible patterns. The following example given in [16] illustrates the probability. Consider a population of size $200.000$, a sparsity of $1\%$ and a target pattern of size 10. The probability that the 10 neurons belonging to the respective pattern become active for a different random pattern is only $9.8 \cdot 10^{-21}$.

For a more detailed discussion of SDRs and their mathematical properties, the interested reader is referred to [16] and [50].

### Temporal Memory and Online Learning

HTM theory proposes that every neuron in the neocortex learns transitions between temporal patterns, tries to predict them and eventually produces motor behaviour. Thus, "everything the neocortex does is based on memory and recall of sequences of patterns." [50] For example, when trying to play a song on the piano, the neocortex moves the finger to the respective keys by predicting the next note based on sensory information about its current position and a stored sequence of notes. The prediction is compared to what actually happens and forms the basis of learning. Consequently, HTM algorithms are designed to work on constantly changing data streams and thus are online learners by nature. In this thesis, the prediction of sequences and the respective learning procedure is leveraged for online anomaly detection on time series data.

### Hierarchical Processing

The human brain is capable of processing constantly changing sensor data of high variety and complexity remarkably well. During a conversation for example, our brain is able to detect and follow another person, despite a constantly changing light situation and movements that lead to a huge variety in the photons hitting the retina.

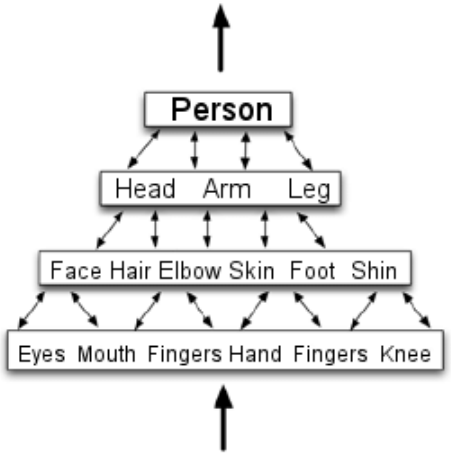



Figure 2.6: Illustration of person tracking with hierarchical processing [43]

The brain is only capable of processing all this information by building an internal representation of the outside world in form of a hierarchy of cortical regions. Figure 2.6 illustrates how hierarchical processing works in the above example of person tracking. A neuron in a lower-level region might fire if it detects an edge of specific size and rotation within its receptive field of view. If the edge leaves the receptive field, it stops firing. By itself, the edge has little to no meaning and could belong to anyone or anything. Only higher-level regions make sense of the input by combining the information given by lower-level regions. Thus, a neuron in a higher-level region would fire constantly when a face is detected, regardless of where it is located. Consequently, higher-level regions in the processing hierarchy are capable of a more general representation of knowledge. For a more detailed discussion around hierarchical processing, especially its feedback mechanism, the interested reader is referred to [51].

Note that this thesis does not use hierarchical processing. Since each region of the neocortex works in the same way due to the common cortical algorithm, current research is trying to understand what is going on within a region before proceeding with hierarchical processing between regions. Therefore there is no practical and only little theoretical work on hierarchical processing yet. However, since it is a core concept of HTM and represents an essential difference to the architecture of the neural network, a short introduction was considered reasonable.

### 2.3.4 A Comparison to Neural Networks

Since both, neural networks and HTM claim to be based on the biological model of a human brain, a direct comparison is inevitable. In both, knowledge is stored through connections, just like in a real brain. In essence, however, a neural network is nothing more than a bunch of neurons hooked together, while HTM theory goes beyond that. Hereinafter, the key differences are discussed.

First, the entire design of the HTM algorithms are derived solely from the human brain. Thus, research does not aim at improving the performance for some specific task, but rather at fully mimicking the brains processing techniques. In contrast, research in the field of neural networks usually tries to achieve a certain behaviour, e.g. to create a work of art or to predict time series. This focus on achieving behaviour by all means has led to a drift away from the biological model. Furthermore, training a neural network often requires a huge amount of data specific to the task at hand, while HTM algorithms are flexible by design. A HTM model can be applied to virtually any task that requires processing sequences without any data being available beforehand; it learns "on the job". Another key difference lays in the type of dataset that is required. HTM algorithms work on dynamic data only, just like the human brain does, whereas neural networks mostly work on static datasets. In addition, neural networks measure success through a global target function that is tailored to the task at hand. Any error is propagated back through the network. Instead, HTM algorithms leverage the nature of dynamic sequence data and compare the predicted values with the actual values and thereby do not require the definition of a suitable target function. The feedback and learning mechanism can be further differentiated in terms of communication. As mentioned in the previous section, the neocortex is organized into a hierarchy of region where each region is performing the same common cortical algorithm. Naturally, for such an organization to work, a huge amount of communication between cells within regions, the regions above and the regions below in the hierarchy is required. Instead, neurons in a basic neural network only communicate in the forward direction during processing time. Some more sophisticated forms, such as a recurrent neural network, do communicate backward, too. Compared to HTM models, this is in some sense a within-layer communication.

In summary, HTM models seem to be a more flexible and dynamic alternative to neural networks, at least in theory, especially when working with time-series data. Consequently, based on the properties of PdM as an AI problem presented in section 2.1.3, HTM promises to be a better choice. Especially the lack of a reliable ground truth and concept drift should be handled well by HTM.

## 2.4 On Extreme Value Theory

In this thesis the EVT is utilised to automatically threshold anomaly scores on data streams. Hereinafter, a brief introduction to the theory will be provided. Initially pioneered in the 20th century, EVT is a branch of statistics that deals with computing probabilities of events that are more extreme than any previously seen, such as tornado outbreaks. As opposed to other methods that find statistical thresholds, the EVT does not make any assumptions about the distribution of the underlying data. This feature makes it very suitable for highly dynamic scenarios where the underlying distribution is often difficult or impossible to determine, such as PdM. Furthermore, any empirical method is not suitable for finding extreme events, since the probability of an unparalleled event equals zero. The EVT solves these challenges by inferring the distribution of the extreme events, regardless of what the original distribution is. In particular, it has been proven that under weak conditions the distribution of extreme events is usually very similar regardless of the original distribution. [91] For example, the distribution of very high temperature values is very similar to the distribution of very high wind speeds, while the distribution of temperature and wind speed values is probably not.

A robust way to fit the tail of the distribution is the Peaks-Over-Threshold (POT) approach, which relies on the Pickands-Balkema-de Haan theorem, also known as the second theorem of EVT. It implies that peaks over a threshold $t$, denoted by $X - t$, are likely to follow a Generalized Pareto Distribution (GPD). With $\bar{F}_t(x)$ being the tail of a distribution $X$, it can be written as

$$\bar{F}_t(x) = \mathbb{P}(X - t > x | X > t) \underset{t \to \tau}{\sim} \left(1 + \frac{\gamma x}{\sigma(t)}\right)^{-\frac{1}{\gamma}} \tag{2.1}$$

where $\gamma$ and $\sigma$ are the GPD parameters shape and scale respectively.

After finding estimates for both GPD parameters, a suitable threshold $z_q$ can be calculated for a batch of observations of size $n$ as

$$z_q \simeq t + \frac{\hat{\gamma}}{\hat{\sigma}} \left( \left( \frac{qn}{N_t} \right)^{-\hat{\gamma}} - 1 \right) \tag{2.2}$$

where $N_t$ denotes the number of peaks within that same batch. Furthermore, the parameter $t$ defines a threshold and $q$ the respective probability, at which it is desired to observe $X < t$. Naturally, $t$ should be set to a threshold that is considered high based on empirical observations of already seen data. [91]

Obviously, the theory as it is given above does not work on streaming data yet. Hence, how to leverage EVT for automatically thresholding a stream of anomaly scores, especially in the context of PdM, will be discussed in section 5.6.

# Chapter 3

# Related Work

## 3.1 On Predictive Maintenance

To provide a comprehensive insight into the research field of PdM, contributions on practical implementations and laboratory experiments are presented in equal measure in the following, leading to a discussion of open questions.

Note that prognostics for bearings is excluded from this review since the respective research mostly uses tools that are very specific to the physical nature of bearings and their degradation process. Consequently, it is not of interest for the rather generalistic approach of this thesis. Instead, the interested reader is referred to [63, 32, 73, 18].

### 3.1.1 Laboratory Experiments

There are numerous contributions that apply AI methods to PdM in a laboratory setting. In [112] for example, the authors achieve considerable results by combining statistical degradation modelling with neural networks to predict the RUL of an aircraft engine. The model was tested on the turbofan engine dataset, which is well-known within the fields of prognostics and PdM. It consists of sensor data recorded during an engine degradation simulation carried out using the C-MAPSS software. [2] Hence, the recorded data was generated entirely through software. Consequently, although achieving good results on the turbofan engine data, it may have its limitations when applied to a real-life setting, where the variety of occurring patterns is significantly higher. The performance of the proposed model was measured through multiple self-defined indicators that are specific to RUL as a regression task, such as a mean error for predictions of RUL values smaller than a self-defined time-delta. Hence, although the model itself may be applicable to

other types of equipment, its performance can only be compared to other regression models. Furthermore, it requires an initial modelling phase specific to the dataset.

Another example for a model developed in a laboratory setting is [67], where the authors leverage regression on sensor data from pumps. The respective dataset consists of sensor data collected over a period of five years from a pump that was operating in a real environment. Hence, the data was not simulated. However, the authors did not publish the dataset, which makes is difficult to reproduce their work. The regression model that was developed works by modelling and predicting each independent variable based on all other independent variables. The corresponding error rates of the bag of models are used as an indication for the health of the pump. An alarm is triggered if the error rate exceeds three standard deviations. Although it works unsupervised, the authors note that it is not able to adapt to a dynamic environment because the individual models are static. The performance was measured by simply checking whether an alarm was triggered before the fault occurred. Such an approach disregards whether the raised alarm was actually related to to the failure and would allow a random predictor to perform well.

### 3.1.2 Real-Life Implementations

Both of the aforementioned contributions developed, trained and tested a model on a static dataset. Other research strives towards an End-to-End PdM solution by incorporating aspects such as data acquisition and model deployment. In [82] for example, the authors developed a solution capable of predicting the RUL of drums within an industrial hot rolling mill. They used a dataset collected from various sources, including sensor and operational data over a period of three years, and successfully integrated multivariate and mixed-type noisy data by conducting thorough feature engineering. As for the ground truth, experts annotated the data with a monthly health state. The degradation process was modelled using a Discrete Bayes Filter (DBF), which is particularly useful for PdM applications due to its ability to adapt to uncertainty and multivariate data of mixed type. However, their solution is not capable to adapt to changes within the processes of the factory. The authors propose to tackle this problem by weighting the training instances, but it would still require to re-engineer the features and to retrain the DBF, leading to a high manual effort. Since the ground truth was available, scoring was done by comparing the root mean squared error of the predictions.

### 3.1.3 Open Issues

The research exemplified in the previous sections does merely intend to be a brief overview. A more thoroughly conducted literature review on data-driven PdM can be found in [111]. Both, the above findings and the mentioned review can be summarized into seven open issues, which are discussed individually below.

**1) Feature Engineering:** The conversion of the raw data stream into meaningful and significant features is either a tedious manual effort or restricted to a very specific type of signal, e.g. vibration. Consequently, additional research should be conducted in the field of automated feature engineering. The work in [33] and [65] already display valuable and partly field-tested solutions for automating feature engineering in the more general domain of industrial IoT.

**2) Interpretability:** Most models, especially DL models, lack interpretability. Consequently, reliability engineering would not be able to reproduce the decision making. Although this may not be directly required from a business perspective, and acceptance within the workforce would increase the adaption rate.

**3) Generalizability:** Another major problem to be solved is the lack of generalizability. Obviously the main reason is the heterogeneity of the machines and the processes involved, which requires individual feature engineering and models. If methods can be found which are easily generalizable, the costs for implementation PdM would decrease significantly.

**4) Supervision:** Most methods in current research are based on supervised learning. Consequently, labelled data is required. However, data from machine failures is by nature hardly available in practice, so unsupervised learning is a promising research area in the future.

**5) Public Datasets:** Overall, research in PdM is held back due to a lack of publicly available data sets. A few valuable datasets are available in [5], but even fewer come from the operating equipment in a real industrial environment. Instead, the data is collected in a laboratory environment and usually does not contain operational data such as process information. Therefore, public data sets are required to enable further research to evaluate AI methods under operating conditions. For example, in a real environment, the sensor readings collected by a machine change with a change in the manufacturing process. Without the ability to adapt to what is considered normal, the previously trained AI model becomes useless.

**6) Comparability:** Another open issue is a lack of comparability, which is a consequence of two challenges. First, data in PdM is sensitive. Consequently, as already discussed in issue 5, there is a lack of publicly available datasets. As a result, researchers often work with data provided by companies, and because the data is business-critical, they cannot publish it. Second, since it is difficult to obtain ground-truth data in PdM, researchers often label the data based on the type of algorithm, for instance using the concept of a prediction horizon in classification. For this reason, researchers tend to develop their own evaluation that is suitable for the particular type of algorithm and task, which makes comparison very difficult.

**7) Setup of an End-to-End System:** An implementation as it was successfully performed in [82] comes with a multitude of additional challenges. Especially the setup of a reliable data acquisition system and the seamless integration of the AI application into business operations can become a significant hurdle and remains an open issue.

**Discussion** To summarize, although a lot of research on solving the problem of PdM with data-driven AI methods has been conducted lately and the performance of the presented models is promising, PdM is still facing a lot of challenges in practical applications. A similar conclusion is drawn by related surveys such as [111], where the main concerns are a lack of available data, a focus on supervised learning methods, a lack of generalizability as well as safety concerns.

## 3.2 On Anomaly Detection

Hereinafter, descriptions of some related work in the area of anomaly detection will be provided. Most relevant are Numenta Platform for Intelligent Computing (NuPIC) and htm.core, the software implementations of the HTM theory with which this thesis works. In addition, other anomaly detection algorithms are discussed, with emphasis on streaming anomaly detection algorithms and anomaly detection algorithms applied in a PdM context.

### 3.2.1 NuPIC

The NuPIC is an open-source software implementation of the HTM theory. It was initially developed by Numenta and first published in 2013. Since then, it has undergone many changes in both implementations as well as concepts. However, by the end of 2017, it was set to maintenance mode due to a change in focus of development efforts at Numenta. Nevertheless, it continues to heavily influence

other implementations of the HTM theory and is the original codebase used for anomaly detection on univariate data streams, which was presented in [17] and from which this thesis is strongly inspired. Therefore, it will be presented here to understand how the HTM theory is translated into a piece of software and leveraged for online anomaly detection. At the time of writing the thesis, the current version is NuPIC v1.0.5, and it consists of a C++ implementation of the core algorithms for efficiency and a Python 2.7 wrapper that allows for experimental code to be developed.

### High-Level Model Design

The HTM model for online sequence learning as it was initially proposed in [35] consists of three main components: An encoder, a spatial pooler and a temporal memory. Each serves a specific purpose in processing a datapoint of a time series. Figure 3.1 illustrates their interaction.

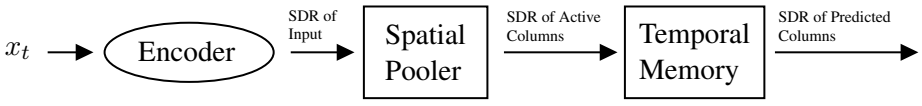



Figure 3.1: Core components of a HTM model

Each data point $x_t$ is converted into a SDR of arbitrary sparsity that encodes the semantics of the input. This SDR is then fed into the spatial pooler, which converts it into a SDR of fixed sparsity and dimensionality that represents the currently activated columns in the HTM system. Finally, a temporal memory uses the currently activated columns to predict which cells will be activated in the next time step and compares the previously predicted cells with the current activation to implement learning. Hence, the input of the HTM model is a data point $x_t$ and the output a SDR resembling a prediction of columns that will be activated in the next time step. Each of the three components is presented hereinafter.

### Encoder

As elaborated in section 2.3.3, all information arriving in the brain must be encoded into a common representation. The cochlea for example, which is part of the inner ear, converts the frequencies and amplitudes of sensed sounds into a sparse set of active neurons. It consists of a set of hairs with each hair being sensitive to different frequencies. When a specific frequency of sound occurs in the environment, the respective hairs stimulate a set of neurons which in turn send the signal into the brain. This biological encoder comes with a few important properties. Each hair

is sensitive to a range of frequencies, and the respective ranges overlap each other. Such a behaviour provides robustness against damaged hairs and also implies that any frequency will stimulate multiple neurons. Consequently, two semantically similar sounds of similar frequencies will have some overlap in the neurons that are activated and their respective semantic meaning is distributed among a set of active neurons, making it tolerant to noise. [80]

This process of encoding sound and its properties is analogous to what is implemented as an encoder in a HTM model. Here, the activated neurons are resembled by a SDR that can be further processed by the spatial pooler. In some sense, the encoder can be seen as the translator between the outside world that is measured in sensory values and the HTM model, that processes information as SDRs. Naturally, the exact design of an encoder depends on the native type of the input data and what semantic characteristics are important for the respective task. In the above example of encoding sounds, for example, higher frequencies might be more important to some animals than to others. Hence, a different encoder design is required. Since Numenta already designed and implemented various encoders that are suitable for a broad range of applications, including time-series anomaly detection, the process of designing an encoder will not be discussed here. Instead, the interested reader is referred to [80].

For the task of time-series anomaly detection, two encoders are required. One to encode time and another to encode scalar values. Both will be briefly presented hereinafter. Note that there is no official paper on either of the encoders. Thus, the following is derived only from the source code published in [10].

**Encoding Time** The time encoder as it is implemented in NuPIC works by combining the encoding of all semantic information that is to be extracted from a date, such as the day of the week, the time of day and whether it is a weekday. For example, take Friday, April 3, 2020, at 14:44. Encoding the day of the week could be as easy as

$$[0, 0, 0, 0, 1, 0, 0]. \tag{3.1}$$

However, such an encoding would prevent any overlap and thus nullify some of the important properties discussed above. To control the semantic overlap, the encoder provides the two parameters width and radius, where width defines the bucket size and radius the desired overlap between two buckets. For a width of three and a radius of two, for example, a Friday is encoded as

$$[0, 0, 0, 0, 0, 0, 1, 1, \boxed{1}, 0, 0] \tag{3.2}$$

and thus would allow for an overlap to a Saturday at the marked position. All other semantic information of the timestamp is processed in the same way, for each of

which a bit array is generated. Finally, all of these arrays are combined into a SDR by concatenation.

**Encoding Scalar Values** Naturally, encoding scalar values is more challenging, because the number of values $x \in \mathbb{R}$ is way larger than for example the number of weekdays. The basic scalar encoder available in NuPIC circumvents that problem by defining the range of allowed values. However, such an encoder is not robust, especially in a streaming scenario where there is no prior knowledge of the incoming data. Consequently, Numenta introduced the Random Distributed Scalar Encoder (RDSE) as a robust alternative. Instead of representing a bucket as consecutive on-bits, the RDSE hashes the bucket and distributes it randomly across the SDR. The semantic overlap can be controlled by the size of the SDR represented by $n$, its maximum sparsity $s$ and the resolution of the encoding, $r$. The latter parameter determines how accurately the scalar values are mapped. For example, for $r = 1$, the scalar values $x = 8$ and $y = 9$ do have an overlap of at least 1 while for any $r < 1$, they would not overlap. Hence, two inputs separated by greater than, or equal to the resolution are guaranteed to have different representations. The following examples shall illustrate the matter. Using a RDSE with $r = 1$, $s = 0.33$ and $n = 9$, $x$, $y$ and $z$ are encoded as follows.

$$x = 8 \mapsto [0, 1, 0, 0, 0, 0, 1, 0, 0] \tag{3.3}$$
$$y = 9 \mapsto [0, 1, 0, 0, 0, 1, 1, 0, 0] \tag{3.4}$$
$$z = 10 \mapsto [0, 0, 1, 0, 0, 1, 1, 0, 0] \tag{3.5}$$
$$\tag{3.6}$$

Note how neighbouring values overlap by 2 and all three values overlap by 1, although $z = 10$ is larger than the size of the SDR.

### Spatial Pooler

Each region in the hierarchical structure of the neocortex has to make sense of highly varying, time-dependent inputs in the form of collective activation patterns of presynaptic neurons. All incoming activation patterns vary in both size and sparsity, and each region has no idea where the input will come from or how many synaptic connections it will involve. For example, the number of connections and the nature of activation patterns from the cochlea is very different from those received by the retina, or by other neocortical regions. Spatial pooling is the process of normalizing these highly varying inputs while retaining the semantic information of each input. Biologically, this is achieved through multiple observed computational principles of the brain. For details on these principles, the interested reader

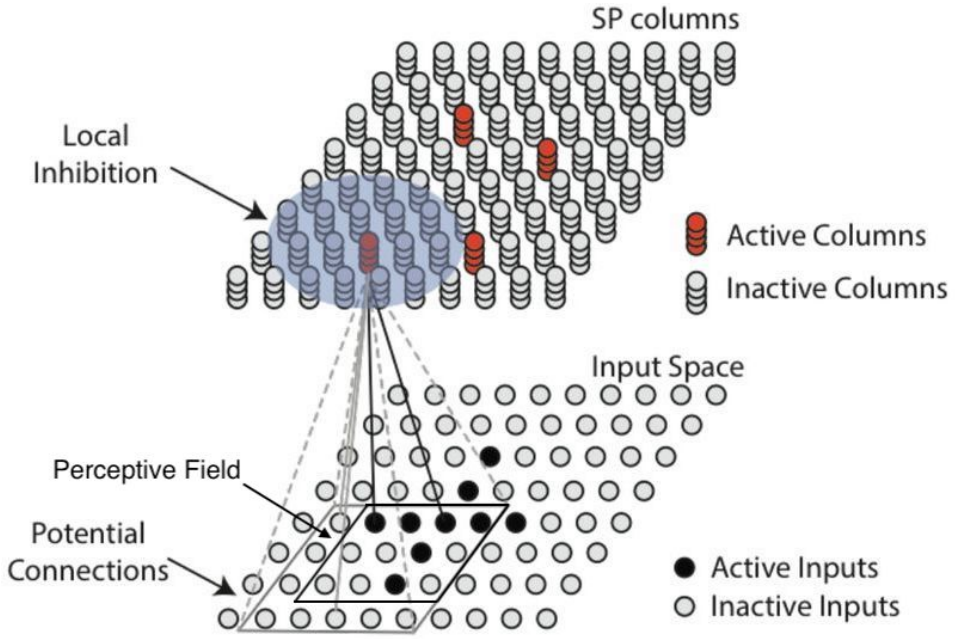



Figure 3.2: Visualization of the spatial pooler (Adapted from [36])

is referred to [36]. By analogy, in a HTM model, the spatial pooler converts input SDRs with an arbitrary sparsity and size into SDRs with a fixed sparsity and size.

Concretely, in an HTM model, the spatial pooler converts the current input SDR into a SDR where each bit in the vector represents the state of a column. Recall that a single neocortical region is represented as a set of columns, each consisting of a set of neurons. Hence, speaking in biological terms, the spatial pooler operates on the level of entire cortical columns. In a full HTM model, the output of the spatial pooler is a representation of the current state and its semantic information, which the temporal memory will then try to predict.

The spatial pooler can be organized into three phases following an initialization. Figure 3.2 visualizes the structure and functionality. The lower layer represents the input space, that in practice is a SDR, which represents the encoded input. The upper layer is a SDR that represents the cortical columns, where each bit defines whether the respective cortical column is active or not.

During an initialization phase, the spatial pooler connects its columns to the individual bits of the input SDR. Each column randomly chooses a set of bits within its potential connections to establish a synaptic connection to. The strength of each synaptic connection is determined by its permanence value. If the permanence of

a connection is larger than a defined minimum permanence threshold, the synaptic connection is considered active. The initial permanence of each newly formed synaptic connection is randomly set to a value within a small range around the minimum permanence threshold, whereby connections in the centre of the area of potential connections are assigned a higher initial permanence.

With the initial connections being established, the spatial pooler now processes each input SDR in three steps. First, it computes the overlap score of each column by counting the number of currently active connections that a single column has with the input. In figure 3.2 for example, the overlap score for the column marked in red is two since it has two active connections, portrayed by the solid lines, with the currently active input bits. Hence, the permanence of these two connections is above the minimum permanence threshold. Further, to diminish insignificant activations, the overlap score is set to 0 if it is below a stimulus threshold.

Next, with an overlap score being available for each column, the spatial pooler selects which column should remain active among neighbouring columns during what is called local inhibition. With a parameter k, which defines the number of active columns per inhibition area, a column remains active if its overlap value is greater than the $k^{th}$ highest overlap value of all columns within its inhibition radius, similar to a $k$-winners-take-all system. Local inhibition helps to keep the sparsity of activation in the SDR relatively constant by ensuring that only a small fraction of the connected columns that receive most input become active.

In a final step, the spatial pooler learns how to adapt to the input stream by applying the principles of Hebbian learning. Hebbian learning is a learning strategy that can be summarized by the term "fire together, wire together". Essentially, it describes the idea of increasing the strength of a connection between two neurons when they are repeatedly activated together. Applied to the spatial pooler, this means that for each column that remains active after local inhibition, all synaptic connections are strengthened, while all synapses of all inactive columns are decremented. Furthermore, the inhibition radius is recalculated as the average size of all perceptual fields across all columns. As illustrated in figure 3.2, the perceptive field is the area of active connections within the potential connections of a column. By coupling the inhibition radius to the perceptive field, columns with a large receptive field and thus potentially many active columns are more inhibited than others, which ultimately leads to a more stable sparsity.

With the design presented above, some notable properties can be ascribed to the spatial pool, which in turn are leveraged by the temporal memory algorithm. For example, it creates a common representation with fixed size and sparsity, utilizes all resources available in the network, is robust to noise, fault-tolerant, and learns online by design. For further details on its properties as well as mathematical formalizations, the interested reader is referred to [36].

### Temporal Memory

The temporal memory algorithm is derived from the biological design of a neocortical region. Specifically, it resembles the lateral synaptic connections between neurons in the same neocortical region which is believed to learn and predict transitions between activation patterns in the network. Hence, the goal of the temporal memory is to learn sequences of activation patterns to predict which neurons will be activated next. As introduced above, the spatial pooler outputs a SDR representing the activation state of all columns within a region. Each column further consists of several neurons that provide a more detailed representational capacity. Moreover, each of these neurons has lateral synaptic connections to other neurons within other columns throughout the region. These lateral synaptic connections depict the temporal context and are leveraged to predict which neurons will be activated next. In summary, the temporal memory algorithm continuously updates the permanence of all lateral connections as well as the state of each neuron to reflect whether it is currently active and whether it is predicted to be activated in the next time step. The entire temporal memory algorithm can be organized into an initialization followed by a three-step process.

During initialization, each neuron within each column is randomly connected to a set of neurons, and each potential synapse is initially assigned a non-zero permanence value. These permanence values are chosen so that some synapses are above the minimum permanence threshold and others are not.

After initialization, the temporal memory can start processing the incoming columnar activation patterns produced by the spatial pooler. First, it computes the activation state of each neuron within each column. For a neuron to become active, the corresponding column must be active. Hence, if a column is not active, none of its neurons can become active. However, if a column is active, the respective neurons can become active in two ways. First, if there are neurons within the column that are currently in a predictive state from the previous time step, these neurons will become active. Alternatively, if no neurons are in a predictive state, all neurons will become active. This process is called bursting and is supposed to simulate the growth of new synaptic connections.

Next, the algorithm computes which neurons to put into a predictive state by counting the number of active synaptic connections to a currently active neuron. A synaptic connection is regarded as active if its permanence value is above the minimum permanence threshold. In other words, a neuron is put into the predictive state if it has a strong connection to a neuron that was activated by the current input.

Finally, all synaptic connections are adapted to the current input in order to implement learning. Very similar to the spatial pooler, the temporal memory algorithm uses Hebbian learning principles and reinforces or decays the permanence

value of the synaptic connections, depending on their contribution to the state. For an active neuron, if it was predicted to become active, any active synaptic connection to the neuron is reinforced by raising the respective permanence value. Otherwise, if it was not predicted to become active, the temporal memory chooses the neuron with the most synaptic input, hence the sum of all permanence values of all connections with that neuron, and increases all permanence values in order to be able to represent that sequence in the future. Lastly, if a neuron is inactive but was predicted, all synaptic connections to that neuron are weakened.

In essence, both the spatial pooler as well as the temporal memory work very similar, just on another level of detail within the HTM model. The spatial pooler learns to normalize the current input and activates columns to represent it, while the temporal memory memorizes the temporal context of a sequence and represents it by activating the individual neurons within a column. For further discussions as well as a mathematical formalization of the algorithm, the interested reader is referred to [35].

### Streaming Anomaly Detection

By design, a HTM model is capable of comparing the actual input to its prediction since both are represented as SDRs. As presented in [17], a measure of how anomalous a given datapoint is can be obtained by simply measuring how many of the predicted columns become active, hence are predicted right. If a majority of the cells that are currently in the predicted state become active, the data point is assumed to be less anomalous and vice versa. Given the current input $x_t$ and its sparse representation $a(x_t)$, the prediction error $s_t$ is obtained by

$$s_t = \frac{\pi(x_{t-1}) \cdot a(x_t)}{|a(x_t)|} \tag{3.7}$$

where $\pi(x_{t-1})$ represents the networks internal prediction of $a(x_t)$ and $|a(x_t)|$ is the total number of on-bits in $a(x_t)$. Put simply, the prediction error is a measure of how many on-bits are overlapping between the input and the output SDR.

However, the prediction error only represents the predictability of the current input. In most applications, the data will be suspect to unpredictable noise. Consequently, instantaneous predictions will be often incorrect. To handle such noise, the authors propose to model the distribution of the error values and use it to check for the likelihood that the current state is anomalous based on the prediction history of the model. The distribution itself is modelled as a rolling normal distribution where, for a window length $W$, the mean and variance are continuously updated

from previous values as follows:

$$\mu_t = \frac{\sum_{i=0}^{i=W-1} s_{t-1}}{W} \tag{3.8}$$

$$\sigma_t^2 = \frac{\sum_{i=0}^{i=W-1} (s_{t-1} - \mu_t)^2}{W-1} \tag{3.9}$$

Finally, the anomaly likelihood is defined as the complement of the Gaussian tail probability $Q$ that is applied to a short term average of prediction errors.

$$L_t = 1 - Q(\frac{\tilde{\mu_t} - \mu_t}{\sigma_t}) \tag{3.10}$$

where

$$\tilde{\mu_t} = \frac{\sum_{i=0}^{i=W'-1} s_{t-1}}{W'} \tag{3.11}$$

and $W'$ is a window $W' << W$, for computing the short term moving average of prediction errors. To report an anomaly, a user-defined threshold is applied to the anomaly likelihood.

Due to the nature of a HTM model, changes in the underlying statistics of a data stream are handled automatically. When a concept drift occurs, the prediction error is initially high, but as the model automatically adapts to the new normal, it then decreases continuously. Another interesting property of the HTM predictor is its ability to predict multiple possible futures at once. As long as the binary SDR vectors are of sufficient sparsity and size, multiple predictions can be presented and evaluated for each incoming data point. The prediction error $s_t$ as it is presented above handles that property gracefully by performing a binary union of each prediction. Hence, if two completely different inputs are predicted, receiving either one will generate a prediction error of 0. Any other input will lead to an error larger 0.

To summarize, leveraging a HTM model for streaming anomaly detection seems to be very promising. Especially in the context of PdM, where the data is very noisy and suspectable to concept drift, HTM should prove superior to other models. However, two major properties hinder the application. First, the model presented above works on univariate data streams only and as discussed in section 2.1.3, the data streams in PdM are multivariate. Hence, an adaption is required that enables the processing of multivariate data streams. Furthermore, in PdM, it is hardly possible to manually set a reasonable threshold, as there is little knowledge of the data, especially in the early stages of a project. Consequently, extending the model by automatic thresholding would significantly increase its usability. A solution to both impediments will be proposed and discussed in chapter 5.

### 3.2.2 htm.core

As mentioned in the previous section, NuPIC was set to maintenance mode by the end of 2017. Although the C++ code is still working, the Python bindings are written in Python 2.7, which was sunset by January 1, 2020. Consequently, for the work in this thesis, another implementation had to be used. There are numerous actively developed options available, such as [34] and [9]. However, to stay as close as possible to the solution that was proposed in [17], htm.core is chosen as HTM implementation. It is an actively maintained community fork of NuPIC and provides an efficient C++ implementation along with Python 3 bindings. The current version is v2.1.15, which is 3711 commits ahead of NuPIC. While continuously improving performance and usability, the project tries to stay as close to the original concept as possible. [77] Consequently, the overall model architecture presented above does not change with htm.core. However, there were some subtle differences in the Application Programming Interface (API) that were not yet understood and resolved, resulting in significantly poorer performance of htm.core when used in anomaly detection. Hence, a comparison of both will be provided in chapter 5 along with a proposal that caused a significant performance improvement of htm.core.

### 3.2.3 Other Algorithms

Naturally, various options are available for the detection of time series anomalies. Although many of these algorithms have the same goal, they impose different requirements on the data and are therefore only applicable to certain applications. As described in section 2.2.4, there are three mutually exclusive classes of anomaly detection algorithms. Hereinafter, a few modern examples will be exemplified for each class with a focus on algorithms that are applied in the context of PdM and work on multivariate data.

#### Batch Learning

In general, there are numerous well-established batch learning algorithms available. One prominent example is the Robust Anomaly Detection algorithm developed by Netflix. [95] As their business naturally demands, it is able to work on highly dimensional data and does not require a normal distribution. At its core, it uses Robust Principal Component Analysis (PCA) which was initially introduced in [27]. It finds anomalies by repeatedly identifying random noise along with a low-rank representation of the data. In PdM, however, batch learning algorithms are rarely used because they only allow retrospective analysis and therefore cannot

be used for prediction tasks. As a result, they are only viable in tools that support decision making by analysing historical failure data. [81]

### Hybrid Learning

Similar to batch learning, a large quantity of algorithms can be classified as a hybrid learner. In principle, any algorithm that requires a training phase falls into this category. For example, the PCA can also be used as a hybrid learner. When the algorithm is applied in PdM, it is first trained on data recorded from a healthy machine to learn its normal state. After training, the anomaly score of an incoming data point is calculated based on its distance to the centre of the transformed coordinate system. In most research, the Mahalanobis distance is used as the distance measure, which calculates the distance to the centre of mass relative to the width of the ellipsoid in the direction of the data point, with the ellipsoid being estimated from the covariance matrix of the training data. [115, 1]

Other research leverages neural networks for anomaly detection. In [96] for example, the authors propose a complex stochastic recurrent neural network for multivariate time-series anomaly detection. The algorithm consists of a recurrent neural network which, once trained, calculates a reconstruction probability for each incoming data point. In order to declare an anomaly, they set a threshold on the reconstruction probability which is automatically calculated by applying the principles of the EVT, introduced in section 2.4, to the anomaly scores of the training set. Consequently, the respective threshold value is static and only adapts to new data after retraining.

Although both models are observed to work well in the context of PdM, they lack adaptability since both must undergo their training offline and do not learn continuously. For instance, if concept drift occurs, they would have to be retrained and there is no automatic way proposed to recognize when and how this should occur. Without any retraining, the model would continuously output false positives.

### Online Learning

Besides the HTM-based model presented in section 3.2.1, there are, at least for the modest knowledge of the author, only a few algorithms for the detection of online anomalies.

First, kernel-based approaches, which have been developed and applied mainly in network-related applications, such as intrusion detection. One example is called EXPected Similarity Estimation, presented in [86], which works on multivariate data streams and obeys all requirements of true online learning algorithms laid out in section 2.2.4, including continuous learning. The authors applied the algorithm

to various real-world datasets, including surveillance and network intrusion data, and the results are promising. Another kernel-based online learning algorithm for multivariate data is proposed in [19]. Although yielding good results in their respective application, in direct comparison to HTM they both perform significantly worse. [62]

Another common approach to real-time anomaly detection is lightweight statistical techniques. Although most of these methods only work on univariate data streams, they will be briefly mentioned here for the sake of completeness. Examples for simple statistical methods include probability tests, changepoint detection, and rules. [54] A popular framework that combines statistical methods into a single tool is Skyline. It essentially works as an ensemble of statistical recognition techniques and evaluates a data point as anomalous if most algorithms agree with it. It is open-source and available at [41]. Although being capable of observing multiple data streams at once, it is only evaluating per-stream and hence operates in an univariate manner. Furthermore, statistical methods are observed to detect spatial anomalies only and can therefore be considered unsuitable for PdM.

However, a statistical real-time anomaly detection algorithm worth mentioning is [91], since they applied the principles of the EVT to detect anomalies in univariate data streams. As opposed to the aforementioned work presented in [96], they employed the proposed algorithm directly on the data stream and not on the anomaly scores. Their algorithms work by declaring extreme values on either static or streaming data, including a variation that handles concept drift. In section 5.6, the proposal of an adaptive univariate anomaly detection algorithm based on the EVT is combined with the work of [96] to develop an adaptive algorithm for automatic thresholding of anomaly scores in an online manner.

## 3.3 On Benchmarks

As already discussed in section 3.1.3, PdM currently lacks a way to compare algorithms. Hence, there is no related work on PdM benchmarks available. Instead, this section will introduce the NASA Prognostics Data Repository, since it contains most of the few public PdM datasets available. In addition, the NAB, a benchmark for streaming anomaly detection algorithms developed by Numenta, is discussed, as the work in this paper was influenced by it.

### 3.3.1 NASA Prognostics Data Repository

The NASA Prognostics Data Repository is a repository of prognostics data sets established and maintained by the NASA Prognostics Center of Excellence. [5] As

of today, it contains a total of 16 datasets donated by various universities and companies, including NASA itself. With its exclusive focus on prognostics, it mostly contains time series data ranging from a nominal state to a functional failure of some system. However, most of the datasets are either from bearings or from small electrical components such as batteries. Since the degradation of such small components is better modelled by incorporating their physical characteristics, they are not really applicable for PdM. Nevertheless, there are some datasets from industrial machinery, which are interesting for the use in PdM. The most cited one is the Turbofan Engine dataset, which consists of sensor data recorded during an engine degradation simulation. Recent examples for related work using data from the NASA Prognostics Data Repository can be found in [74, 96, 108].

Although the data repository offers valuable data sets, it does not provide a way to compare algorithms consistently. Therefore, most of the research done with these data sets develops its own evaluation metrics, which makes comparison difficult. In addition, all data is only provided in a raw format, from text to MatLab files. Hence, any research must first invest in pre-processing and understanding the data to bring it into a usable format before an algorithm can be tested. Furthermore, there is very little data collected by a real-world machine. Instead, most data is either simulated or recorded in a laboratory environment.

Consequently, there is a need for a benchmark that provides a unified way of sharing and comparing algorithms on ready-to-use datasets. Such a benchmark is proposed in chapter 4, thereby adding a significant milestone for open research in the domain of PdM.

### 3.3.2 NAB

The NAB is a benchmark for univariate streaming anomaly detection algorithms that was initially developed by Numenta and is now continued as a community project in [11], similar to the HTM implementation. It is comprised of 58 labelled datasets and a novel evaluation mechanism specifically designed for streaming anomaly detection. The majority of the data is real-world data collected from various sources ranging from cloud server metrics over advertisement clicking metrics to traffic data. Since the benchmark developed in this thesis drew a significant amount of inspiration from NAB, its details will be laid out hereinafter.

#### Datasets

In total, the benchmark contains 58 time series from various application areas, including server metrics, taxi and traffic data and click rates of online advertising. Table 3.1 provides some insights into the datasets. Although some datasets stretch

over almost a year, the 50% quantile is only at 16 days with a time delta between two successive data points of 5 minutes. Hence, the datasets are exceedingly small.

| | length | duration | time delta (in minutes) |
|---|---|---|---|
| mean | 6302.72 | 39 days 08:28:41 | 12 |
| min | 1127 | 4 days 07:30:00 | 5 |
| 50% | 4032 | 16 days 03:46:30 | 5 |
| max | 22695 | 328 days 15:00:00 | 60 |

Table 3.1: Description of the NAB datasets

### Labelling

To evaluate the algorithms, NAB provides ground truth anomalies. While for seven datasets the ground truth anomalies and their real causes were known, all others were labelled by hand using a three-step process. First, three humans tagged timesteps at which anomalies appeared to start. Next, the raw labels of all three labellers were combined into buckets. If a bucket contains at least two labels, it is considered a true anomaly. Finally, the labels were converted into anomaly windows to capture the full anomaly, and not just its starting point. The size of these windows was chosen to be large enough to allow early detection of anomalies without allowing too early detections to count as true positives. Specifically, each dataset was allowed a total anomaly window length of 10% of the dataset length. The length of each anomaly window was defined by the total window length divided by the number of anomalies in the dataset, and the position of each window was set to have its centre at the ground truth anomaly obtained in the previous step.

### Test Process

Overall, the NAB test process consists of three phases. First, all algorithms process each time-series and calculate a raw anomaly score between 0 and 1 for each data point. During processing, no look-ahead is allowed. However, the algorithms do know the minimum and maximum of each time-series beforehand. In a second step, all raw floating-point anomaly scores are converted to discrete values that indicate the presence or absence of an anomaly by retrospectively setting a threshold. Using a simple hill-climbing routine, NAB automatically finds the optimal anomaly threshold for each algorithm, hence the threshold resulting in the best overall evaluation. In a final step, the discrete values are used to calculate a weight for each anomaly detection before combining all evaluations into a final score. The evaluation employed in NAB was developed by Numenta to be espe-

cially suitable for streaming scenarios. Given an anomaly window, an individual detection is weighted based on its relative position to the window $y$ as

$$\sigma^A(y) = (A_{TP} - A_{FP}) \left( \frac{1}{1 + e^{5y}} \right) - 1 \tag{3.12}$$

where $A_{TP}$ and $A_{FP}$ is the true positive and false positive weight of the application profile. The idea of application profiles is to portray the different emphases as to the relative importance of true positives, false negatives, and false positives that different applications may place. NAB defines three application profiles: standard, reward low false positives, and reward low false negatives. During testing, NAB calculates the evaluation for all three application profiles on all datasets for all algorithms. Figure 3.3 exemplifies how the formula presented above is applied to calculate the weight of individual detections given an anomaly window and the relative position of each detection. In the given example, the first point is evaluated as a false positive. Within the window, there are two detections, of which only the first is counted for the true positive weight. After the window, both detections are evaluated as false positives and weighted according to their distance to the previous detection window.

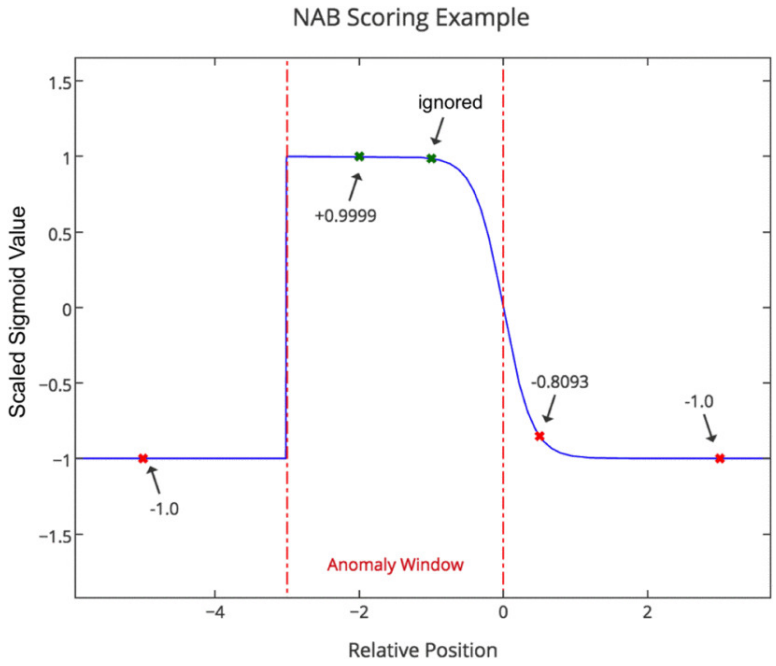



Figure 3.3: Example of NAB scoring system [62]

After each individual detection is weighted, the evaluation score for a time-series is calculated as the sum of the individually weighted detections plus a penalty for any missed window.

$$s_d^A = \left( \sum_{y \in Y_d} \sigma^A(y) \right) + A_{FN} f_d \tag{3.13}$$

Finally, the raw benchmark score for a given algorithm is simply the sum of all individual scores over all datasets.

$$s^A = \sum_{d \in D} s_d^A \tag{3.14}$$

To improve comparability, NAB normalizes the raw score based on a perfect algorithm, $s^A_{perfect}$, that outputs all true positives and no false positives and a null algorithm, $s^A_{null}$ that outputs no detections.

$$S^A_{NAB} = 100 \cdot \frac{s^A - s^A_{null}}{s^A_{perfect} - s^A_{null}} \tag{3.15}$$

Hence, the final NAB score is a value between 0 and 100, where 100 is the maximum score that an algorithm can achieve.

### Open Issues

As already mentioned above, there exist some open issues with the current version of NAB and since Numenta stopped to actively maintain the repository, it can be assumed that they are going to stay. Although there is an actively maintained community fork of the benchmark, it mainly focusses on adding new algorithms. The work in [92] and [60] already discusses some valid concerns which will be listed below together with some personal observations.

1. Although the labelling process is well described, it is still based on human labellers and is therefore, at least to some extent, arbitrary. Furthermore, the labellers are data analysts and not domain experts in the particular domain from which the dataset originates. For example, a reliability engineer could use his knowledge of the respective machine to better find anomalies. Naturally, such a procedure would fail for multivariate data, since a human cannot reliably comprehend the dynamic correlations in multiple data streams.

2. As already noted in [92], the scoring function presented in equation 3.12 does have a few gaps. First, it is unclear how $y$, which denotes the relative position to the current anomaly window, is defined and what range of values it can take. Next, it remains unknown why the value of $5$ is chosen in $5y$ and what impact this scaling factor has on the overall NAB score.

3. The included datasets are exceedingly small, which is unrealistic, especially in streaming scenarios. Furthermore, due to the small sizes, an algorithm often does not have enough time to adapt to a concept drift.

4. During the test process described above, the algorithms only output an anomaly value between 0 and 1, but not a discrete value indicating whether the current data point is abnormal or not. Such a design is not desirable for some applications, such as PdM. In addition, NAB assumes that a perfect threshold can easily be found in practical application. In reality, however, it can be difficult to find a good threshold for anomaly scores, especially in dynamic environments.

5. The optimization algorithm that finds the optimal threshold on the raw anomaly scores uses local hill-climbing, which does not guarantee to find the global maximum and might therefore underestimate an algorithm.

6. Although there is no look-ahead allowed for the algorithms, they know the minimum and maximum value that the incoming data can take. Numenta argues that this is realistic for most real-life scenarios and although it may be valid for univariate processing, it certainly is not for multivariate streaming data and highly dynamic environments.

7. During testing, NAB calculates the evaluation for all three application profiles on all datasets for all algorithms. Hence, all datasets are evaluated in the same manner, regardless of their context. It would be more realistic if the application profiles were assigned to each dataset individually, depending on their respective application.

8. From a technical perspective, the NAB framework does have some minor limitations, too. Its closed architecture prevents any use other than to test algorithms on the entire benchmark. For example, some researchers might be interested in checking the results of their algorithm on specific data sets. In addition, the framework writes the valuations only to the file system, thus preventing parallel execution and making it more difficult to implement parameter optimization.

To summarize, NAB can serve as a great inspiration for a benchmark, although especially the scoring function and the technical architecture should be taken with a grain of salt. In the design of the prognostics benchmark presented in chapter 4, both, shortcomings and virtues of NAB, are considered.

# Chapter 4

# Prognostics Benchmark

Motivated by a lack of comparability in PdM and its superordinate field prognostics, the following chapter introduces a novel prognostics benchmark that provides a unified way of sharing and comparing algorithms on ready-to-use datasets, thereby adding a significant milestone for open research. Although the focus will be on PdM, its open design enables researchers to also contribute datasets and evaluation measures from prognostics.

First, the previously conducted research on PdM and benchmarks is summarized into a list of requirements that underlie the design of the proposed benchmark. This is followed by a description of the design and technical architecture of the benchmark, including details of its core features and possible use cases. Finally, the content currently contained, consisting of data sets, algorithms, and a new type of evaluation measurement especially designed for PdM, is presented.

## 4.1 Framework

Any benchmark mainly consists of datasets, algorithms, an evaluation measure suitable for the problem at hand, and a framework that enables working with and adding to all three. Therefore, the prognosis benchmark is a framework that defines a uniform way of storing and interacting with data sets, implementing algorithms, and evaluating them in the field of prognostics. To achieve the overall objective of a general establishment of the benchmark within the research community, the following two principles must be respected.

1. **Open Architecture:** One design flaw of NAB is its closed architecture. Users can only test algorithms on the entire benchmark and the results are only accessible from the file system. Instead, the prognostics benchmark

must have an open architecture, giving anyone full access to all components on every abstraction level and thereby allowing more flexible use.

2. **Ease of Use:** To increase adaption, the benchmark must be developed in Python 3.7, which is the most-loved language for data science according to the recent Stack Overflow survey. [6] Furthermore, for an ease of use, it must provide utility functions wherever possible. For instance, the analysis of results should be supported by plotting capabilities, and the implementation of algorithms should be assisted by various utility functions, e.g. the conversion of timestamps into RUL values. In addition, it must allow sophisticated use-cases such as parallel processing and parameter optimization.

### 4.1.1 Requirements

Inspired by the shortcomings and virtues of NAB listed in section 3.3.2 and by the properties of PdM derived in section 2.1.3, the requirements for each of the components is presented hereinafter.

#### Datasets

Meaningful data is the basis of every ML project. Only if the data represents the problem that one is trying to solve accurately, the trained model outputs meaningful predictions. Similar, without significant data of good quality, a meaningful comparison of algorithms is not possible and a benchmark becomes useless. However, for benchmarks, the problem statement is much broader as for the most ML projects. For instance, a ML project may try to decrease maintenance cost for a very specific type of equipment, while the benchmark aims to compare algorithms on the entire domain of PdM. Thus, there is usually more data of higher variety required. Consequently, to be able to compare algorithms on multiple different datasets, a benchmark has to provide a unified way of storing and accessing them. Especially in PdM, where both the semantic and syntactic variety among datasets is huge, such a scheme has to be designed with great care. Another challenge is labelling. Without a ground truth, the performance of an algorithm cannot be measured. As discussed in section 2.1.3, providing a ground truth in PdM is particularly difficult.

The aforementioned challenges together with the properties of PdM and the learnings from examining NAB can be combined into a list of requirements defining how datasets are to be handled within the benchmark.

1. **Unified Data Model:** To abstract away the high semantic and syntactic diversity among datasets, the benchmark has to provide a unified data model

that any data can be transformed into.

2. **Failure Labels Only:** In PdM, the relationship between patterns preceding an equipment failure and the failure itself is often unknown. Some researchers solve this by manually labelling the data using expert knowledge. However, such a procedure is very costly, and for public datasets often impossible. Similar, NAB hand-labelled the datasets using a well-defined procedure. While this may work for univariate data, it is practically impossible for multivariate data. To circumvent these challenges, the prognostics benchmark should not use any labelling other than the times at which a machine failed, as this information is usually available.

3. **Minimal Pre-Processing:** Since PdM mostly works on streaming data, any pre-processing must be applied directly to the data streams. Some examples of streaming feature engineering can be found in [57, 88]. To mimic that challenge in the benchmark, the pre-processing of included datasets is restricted to the aforementioned transformation of non-numeric values. There may be some cases where additional pre-processing makes sense, but this is to be handled in a case to case manner. Consequently, any further pre-processing that some algorithms may require is to be applied in a streaming manner within the implementation of the algorithm itself.

4. **Sufficient Dataset Size:** One item of review in NAB was the small dataset sizes which are unrealistic and may prevent an algorithm to adapt to a concept drift accordingly. In PdM, there is usually a lot of data being recorded. Hence, the datasets must be of sufficient size. There may be some cases where a small size can be allowed due to the specific application of the dataset, but again, this has to be decided on a case to case basis.

5. **Variable Datasets:** To represent the high variety of machines encountered in PdM, the provided datasets must span multiple machine types.

6. **Well Documented:** As discussed in section 3.1.3, finding suitable datasets is a major open issue in PdM research. Hence, all datasets added to the benchmark must be well documented in terms of source, application, and pre-processing applied.

### Algorithms

Since the main goal of a benchmark is to provide a common method for sharing and comparing algorithms for a problem, adding an algorithm must be as simple and intuitive as possible. At the same time, however, it must be ensured that all

algorithms conform to a set of requirements that the respective problem imposes on them. In NAB, for example, all algorithms must be implemented so that they meet the requirements of a streaming anomaly detector as presented in section 2.2.4. Consequently, the framework must find a balance between ease of use and fairness.

Based on the challenges described above and the properties of PdM, the following requirements can be defined.

1. **Streaming Processing:** To reflect reality, all data must be processed by the algorithms in a streaming manner. Hence, no look-ahead is allowed and there is no upfront knowledge of the data given. As mentioned before, even the pre-processing, if required, must be handled within the algorithm. Note that this requirement does not impose the properties of a streaming algorithm. Instead, just the processing of data must be done in a streaming manner. The algorithms are still allowed to save data and train models.

2. **Full Flexibility regarding the Type of Algorithm:** As opposed to NAB, the prognostics benchmark does not compare algorithms of a specific type, but instead compares any kind of algorithm on a specific problem. Consequently, another requirement is full flexibility regarding the type of algorithm that is being used. Whether the underlying model is based on anomaly detection, regression, classification, or whether it is a neural network or just a statistics-based model must be irrelevant.

3. **Unified Output:** To be able to compare the algorithms considering the previous requirement, the benchmark has to define a unified output type that each algorithm has to return while processing the data. The output should be meaningful in the context of predicting a failure and should not privilege a specific type of algorithm.

4. **Parameterless:** As mentioned in section 3.1.3, generalizability is one open issue of PdM. To incorporate generalizability into the design of the benchmark, all parameters have to be set on a benchmark level. For example, if an algorithm has to manually set a threshold for the anomaly score, it must use the same threshold for the entire benchmark, which may lead to worse results due to the variability of machine types. Consequently, a researcher has to find good default parameters with which the algorithm generalizes well across different datasets.

### Evaluation

In most benchmarks, the type of evaluation that is being used is set by the framework itself. However, this is only useful for domains where a common evaluation measure is established within the research community, such as object detection. [31] Since this is not the case for neither the field of prognostics nor PdM, the benchmark should not be biased about how to evaluate. Instead, it should provide a flexible evaluation framework that allows researchers to make contributions and discuss metrics, in the hope of one day establishing one.

Based on this principle and the learnings from examining NAB, the following requirements can be defined.

1. **Dataset-specific Parameterization:** As elaborated in section 2.1.3, PdM always has a lead time which is defined as the time before the failure that is required for spare part management and scheduling activities. In practice, the lead time is determined for each type of equipment, since the required spare parts are different. Consequently, it must be possible to set the lead time separately for each equipment type, together with other parameters that may influence the evaluation. In NAB, the difference between the applications is implied by the application profiles that were applied to the entire data set. While this is reasonable for the more general application of anomaly detection, the parameters should be set individually in a specific domain such as PdM.

2. **Flexible Evaluation Framework:** As already elaborated, a benchmark should not be biased about how to evaluate if there is no established evaluation metric in the respective field of research. For example, although NAB proposes a novel evaluation metric specifically for the domain of streaming anomaly detection, it is not yet fully established within the community. This is due to the fact that the implementation is not only fixed, but also intervenes deeply with the benchmark itself, so that it can neither be improved nor its code analysed. For instance, other researchers may be interested in trying out how the scale of the sigmoid function influences the overall evaluation score. The technical design of the benchmark makes this impossible. Consequently, the benchmark proposed in this thesis must implement a flexible framework around the calculation of evaluation scores so that existing evaluations can be easily discussed and new evaluations easily implemented.

### 4.1.2 Design

#### Data Model

The main pillar of solving the requirements listed above is the unified data model. Fortunately, a very similar problem has been solved before by software solutions for asset management and plant maintenance. With a lot of simplification and minor adaptions, their data model can be used for the prognostics benchmark, too. Figure 4.1 presents a diagram of the entities and their relationship along with an example using the Turbofan Engine dataset from the NASA Prognostics Repository.

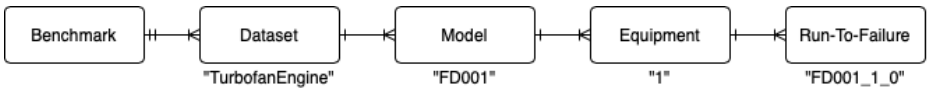



Figure 4.1: Unified data model of the benchmark

Naturally, the benchmark consists of multiple datasets, each uniquely identifiable by a human-readable name. The benchmark does have at least one dataset, while each dataset is part of the same benchmark. To distinguish between different types of equipment within the same dataset, each machine is assigned a model that defines its type. In the Turbofan Engine dataset, for example, the machines were simulated under four different conditions, identified by FD001, FD002, FD003 and FD004 respectively. Hence, the dataset contains four different types of equipment. Finally, an equipment is the instance of a model, thus the representation of the physical machine. Each equipment has exactly one model, while there is at least one equipment of each model. For instance, the turbofan engine dataset contains around 100 uniquely identifiable machines for each of the four models. Finally, a Run-to-Failure (RtF) is a dataset consisting of time series data recorded from a single machine ranging strictly from a nominal state to a failed state of the machine. For each equipment, there is at least one RtF, and each RtF belongs to one equipment. To store the data in the correct order, all RtFs of an equipment are assigned an index. Therefore, within a dataset, a RtF is uniquely identifiable by its model, the equipment and its index.

The concept of a RtF as a unified way of storing the time series data comes with two major benefits. First, since each time series stored within a RtF ends with a failure state, it implicitly holds the labelling. Second, it does not enforce how a failed state is defined but delegates that responsibility to the contributor of the dataset. Figure 4.2 depicts how datasets may define a failure differently.

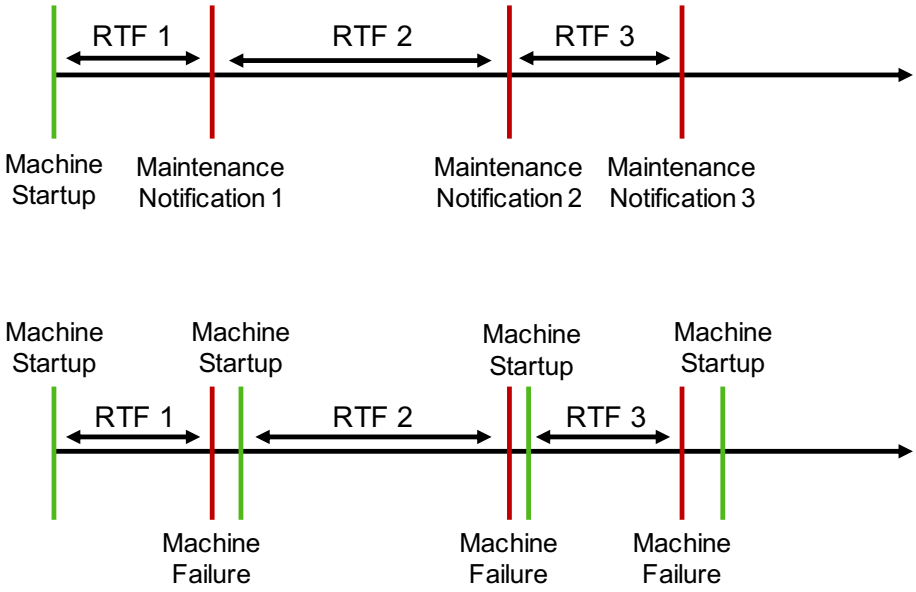



Figure 4.2: Different definitions of a failure

In some cases, a machine failure is represented by a maintenance notification. In standard maintenance software, a notification is used to notify the maintenance department about the abnormal behaviour of a machine. It usually contains the time of detection and a short description of the observed maloperation. Hence, in such a dataset, a RtF ranges from either the machine start-up or the previous maintenance notification to the next one. Other datasets may include a recorded or estimated time for which the machine is in a breakdown state, where they are at least not fully operational. In these cases, a RtF ranges from the time at which the machine leaves the breakdown state to the next failure. Yet another possibility is applications where a machine is replaced after a failure. Here, there is only one RtF for each machine, but probably a lot of machines for each model.

Beside the unified data format, the benchmark also enforces specific data types. As discussed in section 2.1.3, the data in PdM is multivariate and of mixed type. For instance, sensors record the physical condition of the machine, while a process control system provides information about the type of material that is currently being processed. Naturally, such a variety of data sources leads to a large variety of input types. Combined with the requirement of no look-ahead imposed on the algorithms, implementing an algorithm would require a lot of unnecessary effort for dynamically handling different data types. Consequently, to ease the implementation of algorithms, the benchmark enforces numeric values on the input

vector. Note that this does not prohibit categorical values but rather enforces the transformation of non-numeric data during pre-processing. Furthermore, in real-life scenarios, all data used in PdM is time-series data. Consequently, although many of the public datasets only have an integer index, the benchmark enforces real timestamps. Hence, any non-timestamp index has to be converted into a real timestamp during pre-processing. Again, the enforcement of real timestamps is not only more realistic, but also facilitates the implementation of the algorithms, since they do not have to deal dynamically with multiple types of indexes.

Naturally, the design specified above requires certain pre-processing to be applied. Especially since the open definition of a failure imposes a great responsibility on the contributor of a dataset, any dataset that is being added to the benchmark must be well documented. Specifically, the benchmark enforces information about the dataset source, a short description of the use case, and optionally its license. Furthermore, in order to be fully transparent and reproducible, the script used to pre-process the raw data into the format required for the benchmark must also be contributed directly to the repository.

### Algorithms

Based on the requirements listed before, the main challenge of designing a framework for the implementation of algorithms is to allow full flexibility regarding the type of algorithm while ensuring comparability. In the benchmark, this problem is solved by enforcing a unified output on the algorithms. Specifically, all algorithms in the benchmark must output a Boolean value that indicates whether or not an alarm should be triggered at the current time. The idea behind that is based on the following observations.

First, as defined in section 2.1.3, PdM attempts to predict a functional failure to plan appropriate maintenance work and prevent the failure. In practice, a reliability engineer would create a maintenance notification to trigger the maintenance process based on the feedback from the algorithm. Essentially, the algorithm informs the engineer by triggering an alarm indicating that the machine should be repaired as soon as possible. Hence, at least from a practical perspective, enforcing a Boolean value as output can be considered reasonable.

Next, although different algorithms are used in PdM research, they all essentially try to provide information about whether the machine should be maintained. For example, based on the RUL and the lead time, a reliability engineer can decide whether or not to create a maintenance notification. The decision can be as simple as checking whether the predicted RUL is smaller as the lead time. Similarly, a classifier supports a reliability engineer by indicating whether the machine will fail in the next $n$ minutes. Here, the decision is made based on the class that

the algorithms predicts. Furthermore, unsupervised anomaly detection algorithms support the decision process by indicating how abnormal the machine currently behaves. In practical applications, reliability engineers either directly observe the live stream of continuous anomaly scores or set a threshold that converts the continuous anomaly scores into discrete values indicating whether or not maintenance should be triggered. The latter is also used in NAB, although the conversion is done retrospectively by finding the anomaly threshold that leads to the best possible score. However, since the benchmark does not allow any look-ahead, an unsupervised anomaly detection algorithm must set the appropriate threshold either benchmark-wide or dynamically based on the data already seen.

#### Evaluation

Last but not least, a framework for the implementation of evaluation metrics must be established. Naturally, the core functionality of any evaluation is to compare the output of an algorithm with the ground truth stored in the data. In the prognostics benchmark, the ground truth is implicitly stored within a RtF, and the algorithms output a discrete value for each datapoint indicating whether or not to raise an alarm. Therefore, for each RtF it must be assessed how well the algorithm was able to predict the failure based on the alarms it has triggered. These individual assessments must then be combined into one evaluation for each data set and a final score for the entire benchmark. Of course, the individual RtF evaluations must take business requirements such as the lead time into account, which are defined for each dataset. For example, if the maintenance of a machine is triggered after the lead time has expired, a failure can no longer be prevented and the respective evaluation should be low.

Although the design of the evaluation framework feels natural when viewed from a PdM perspective, creating an appropriate evaluation metric within this framework presents a number of challenges. Nevertheless, a suitable metric will be proposed in 4.2.3, alongside a discussion about the respective challenges.

### 4.1.3 Implementation

The benchmark itself is developed in Python 3.7. Besides being the most-loved programming language in data science according to the recent StackOverflow developer survey, Python comes with a number of functionalities useful for developing a benchmark, such as multiprocessing, advanced data processing, and sophisticated ML and DL capabilities. [6] Hereinafter, the implementation of the benchmark is presented, with a focus on the details relevant to the research community. In particular, it will be explained how the data is handled, how new data

sets can be added, how the test process is realized, how new detectors and evaluators can be implemented and what additional functionalities are included to support the user in working with the benchmark.

Figure 4.3 presents a high-level Unified Modelling Language (UML) diagram of the final benchmark architecture. Of course, the functionality of all classes is explained in the following sections, but if the reader wishes a more detailed graphical representation, he or she is referred to appendix A.

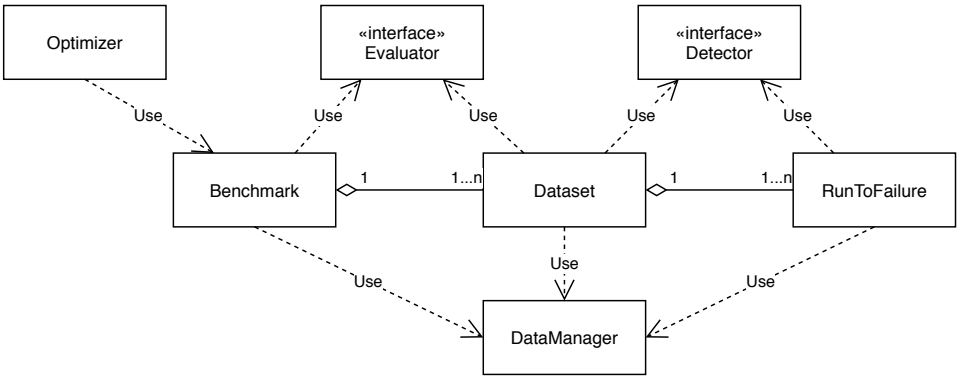



Figure 4.3: High-level UML diagram of the benchmark

### Data

Based on the requirements and design discussed above, the benchmark is required to enable the user store new datasets in the respective format and to easily access the stored data in a meaningful manner. At first, the idea was to use a MongoDB to efficiently store and access all data. However, this would hurt the ease of use of the benchmark, since an user would be required to install the database. Also, the data that would be contained in a database dump could not have been directly shared in the repository due to the file size limit of 100 megabytes imposed by GitHub. As a result, the data is stored in smaller comma-separated values (CSV) files, one for each RtF. While this limits the size of the data stored in a RtF to 100 megabytes, this limit was not nearly reached with the datasets currently contained in the benchmark. To add to or access the stored data, a DataManager is available which acts as a semantically enhanced database client and contains all basic methods to read from and add to the datasets. A detailed UML diagram of the DataManager can be found in appendix A.

To add a dataset to the benchmark, the contributor has to provide a detailed documentation including the source of the data, a brief description of the use case, and a pre-processing script written in Python 3. To ensure reproducibility, the

script must include all processing steps required to convert the data from the raw format in which it is accessible through the given source into the format enforced by the benchmark. To write the data to the benchmark, the contributor must use the respective functionalities provided by the DataManager. Of course, the format of the data is checked by the DataManager before writing to the file system.

To work with the data, the user should not be directly using the database client but instead a set of classes that provide the functionalities expected from a benchmark. Based on the relationship diagram presented in figure 4.1, a possible architecture would be to represent each of the entities with one class. However, since the classes Dataset, Model, and Equipment would hold the same functionalities just on another level of detail, e.g. query models, query equipments and query RtFs, Model and Equipment is abstracted away. An instance of the Dataset class thus contains all methods required to access the data within the respective dataset. However, the time series data itself is accessed via an instance of RunToFailure. The reason for this lays in the implementation of the test process and will be explained in the following section. Finally, an instance of Benchmark can be used, inter alia, to query information about available algorithms, evaluators and data sets. It is important to note that both Dataset and RunToFailure can be used independently of the Benchmark class for maximum flexibility. A user can simply create an instance of Dataset or RunToFailure based on the respective identifier and work directly on the data. Again, for more information on available methods, the reader is referred to appendix A.

### Evaluator

Based on the design described above, an implementation metric must essentially provide three methods. First, it must assess how well an algorithm predicted a failure based on the alarms that it raised for a RtF. Hence, the respective method receives a series of timestamps with Boolean values indicating when alarms were raised and returns a single scalar value. Next, all RtF evaluations of a dataset must be combined into one scalar evaluation. Here, the method takes a list of RtF scores and returns a scalar value. Finally, all dataset scores must be combined into one benchmark score. Again, the method takes a list of dataset scores and returns a single evaluation value. Essentially, the design of the evaluation is very similar to the divide-and-conquer approach usually used in sorting procedures, since the evaluation at benchmark level is divided into an evaluation at dataset level, which in turn is divided into an evaluation of individual RtFs. However, the combination to a final score can be adjusted individually for each level.

To ensure that all evaluation metrics adhere to the aforementioned implementation, they must subclass the abstract class Evaluator. Besides providing utility

functions, it enforces the implementation of the three methods mentioned above. To enable parameterization on a dataset level, an evaluator holds a static object containing the respective configuration for every dataset in the benchmark. Upon initialization, it establishes the correct parameters depending on the given dataset identifier. Note that all parameter sets must at least contain a lead time parameter. Figure 4.4 depicts an UML diagram of the Evaluator class.

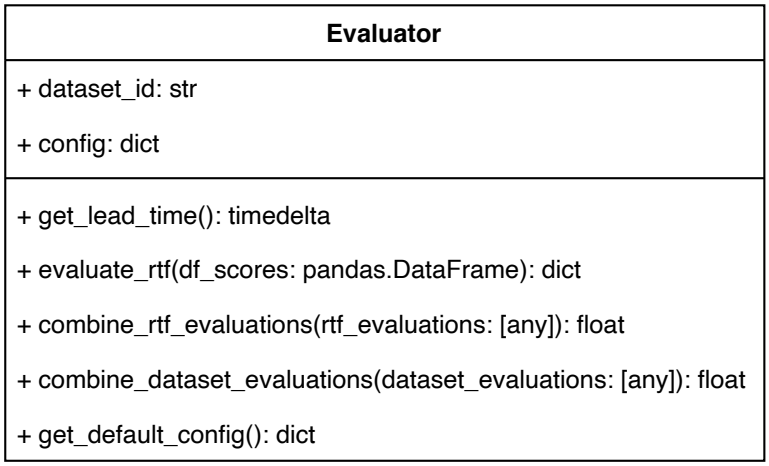



Figure 4.4: UML diagram of the evaluator base class

### Detector

Algorithms are implemented similarly to evaluators. Again, they must subclass the abstract class Detector, which, besides providing utility functions, enforces the implementation of a method that must return a Boolean value indicating whether or not to raise an alarm based on the current timestamp and feature vector. Furthermore, it can implement a method that is called when a failure is reached e.g. to train a model based on already seen data. For convenience, the respective method receives the last RtF object, including the time-series data. Figure 4.5 shows a UML diagram of the Detector class.

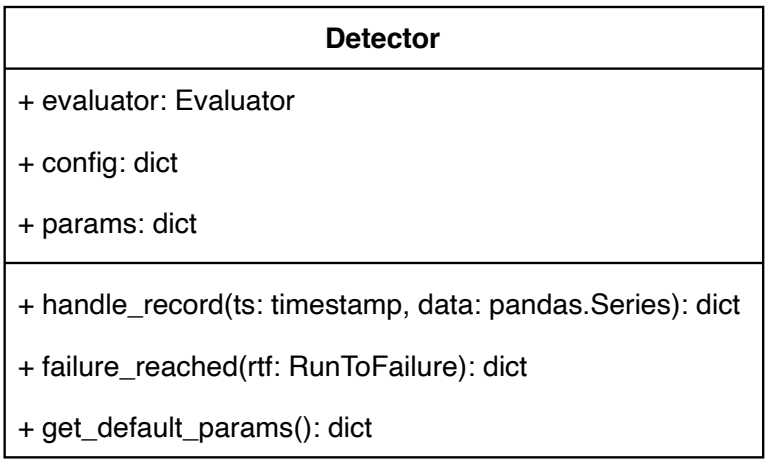



Figure 4.5: UML Diagram of the detector base class

Note how this design does not restrict the type of algorithm to online learners. For example, if a user wants to implement a supervised regressor, he or she can use the failure reached method to store the RtFs, apply the RUL, and train the model. Note that the detector holds an instance of the evaluator class and therefore has access to the lead time parameter. Naturally, while the first RtF is processed no model is trained yet and false must be returned for every data point. A more detailed explanation of how different types of algorithms can be implemented will be given in section 4.2.2.

### Test Process

Naturally, the test process is the core feature of the benchmark. It combines all of the elements described above to evaluate a given algorithm based on a given evaluator. Figure 4.6 displays a high-level overview. Only one simple method is used to trigger the entire test process, which is exposed by the Benchmark class. Apart from some optional parameters that allow for customization, only the names of the detector and the evaluation class are required. Internally, the method checks whether the given classes are available within the benchmark and dynamically loads them. After that, for each dataset currently included in the benchmark, a process is spawned using the multiprocessing capabilities of Python. Within each process, the given algorithm is evaluated on the entire dataset. Finally, all individual dataset evaluations are combined into a final evaluation score using the corresponding static method on the evaluator class.

A deeper level of detail is shown in figure 4.7, which is a sequence diagram

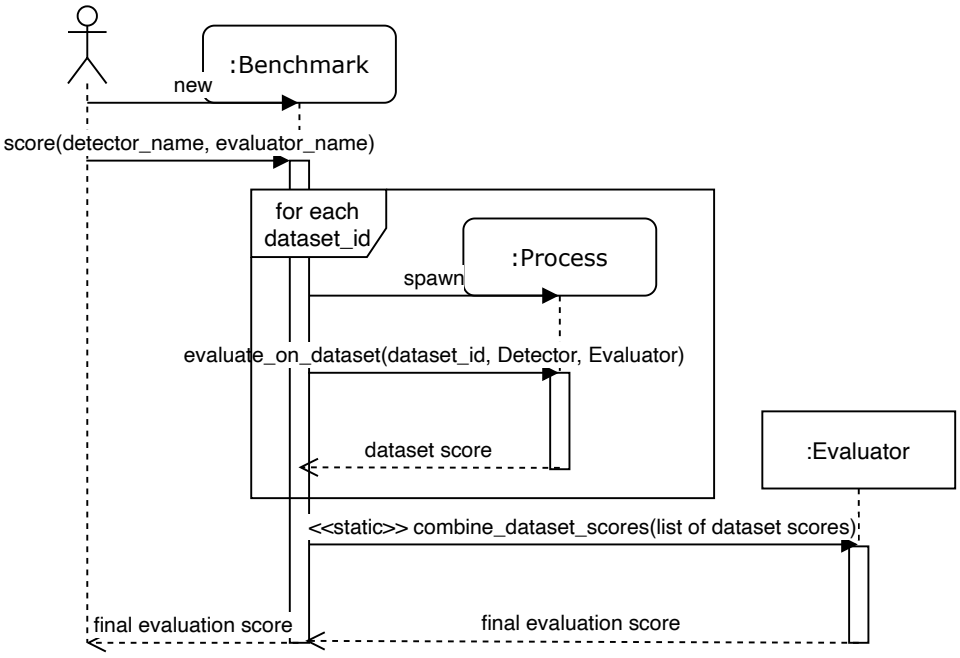



Figure 4.6: High-level sequence diagram of the overall test process

depicting how a given detector is applied to a data set and evaluated with a given evaluator. First, a representation of the dataset is initialized using the respective identifier. In addition to the methods used to query information about the dataset, it exposes two methods that encapsulate the test procedure. One, used by default, creates one detector for each model, while the other creates one detector for each equipment. The decision to evaluate on models by default, hence use the same instance of an algorithm for all equipments of the same type, is based on personal practical experiences as well as reports of practical implementations of PdM, such as [112, 67, 105, 56].

Within this method, an instance of the given evaluator class is created. By passing the current dataset identifier the evaluator is initialized with the correct configuration parameters. Next, an instance of the given detector class is initialized for each model. Finally, the method queries all RtF identifiers for the current model and evaluates the detector for each corresponding RtF, computing an evaluation score for each RtF. The individual evaluations are then combined into a dataset score using the respective method of the evaluator.

Naturally, the processing of the models could be done in parallel to further increase the processing speed. However, for data sets with many models, this usually

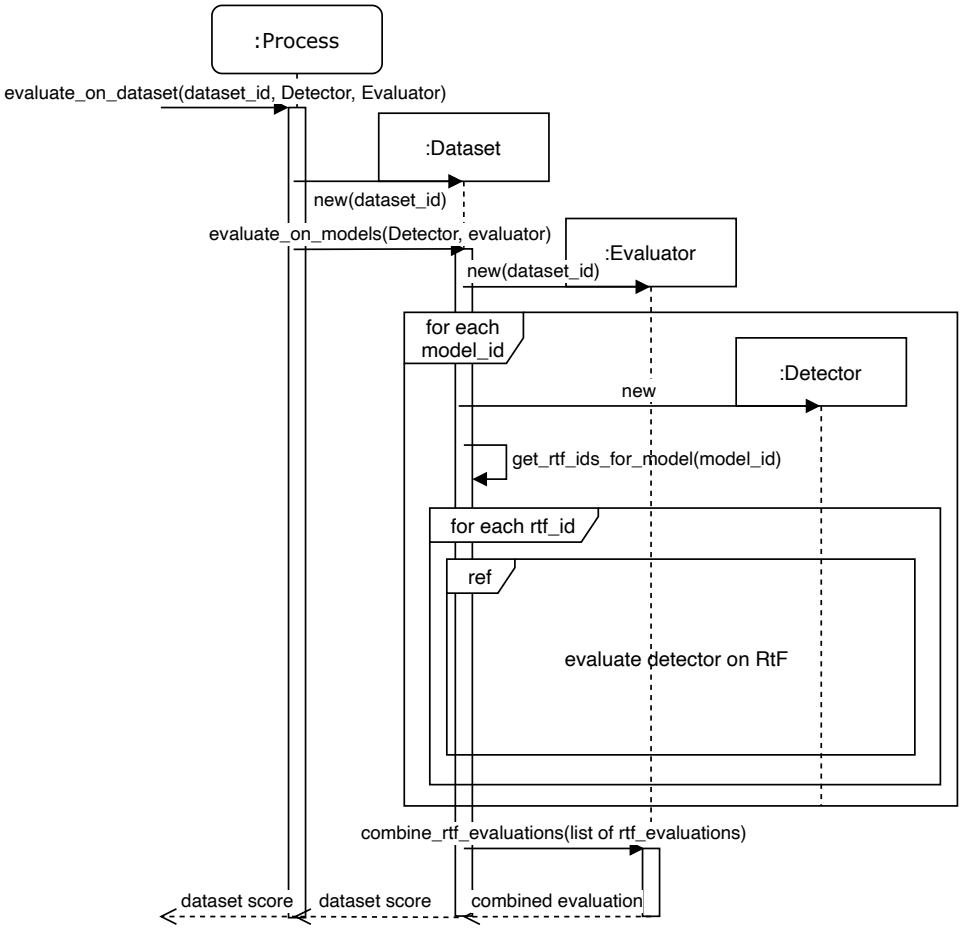



Figure 4.7: Sequence diagram of the test process for a dataset

leads to a memory leak. Consequently, the option for parallel processing has been removed.

It should also be noted that although each RtF is stored in the chronologically correct order, this applies only at the equipment level. Therefore the detector does not process the RtFs in the chronologically correct order, but equipment by equipment. Within each equipment, however, the RtFs are in the correct order. This design was chosen because it drastically reduces the complexity of the data storage design and the implementation of an algorithm. If all data were stored, accessed, and processed in the chronologically correct order, the DataManager would have to query across multiple files, which is very inefficient. Furthermore, an algorithm

would have to implement some kind of model manager that controls which data point is processed by which model. Since usability and adaptability are the main objectives of this benchmark, the simpler design was favoured.

Finally, how the detector is applied to and evaluated on a RtF is depicted in figure 4.8. At first, a representation of the RtF is initialized given the current iden-

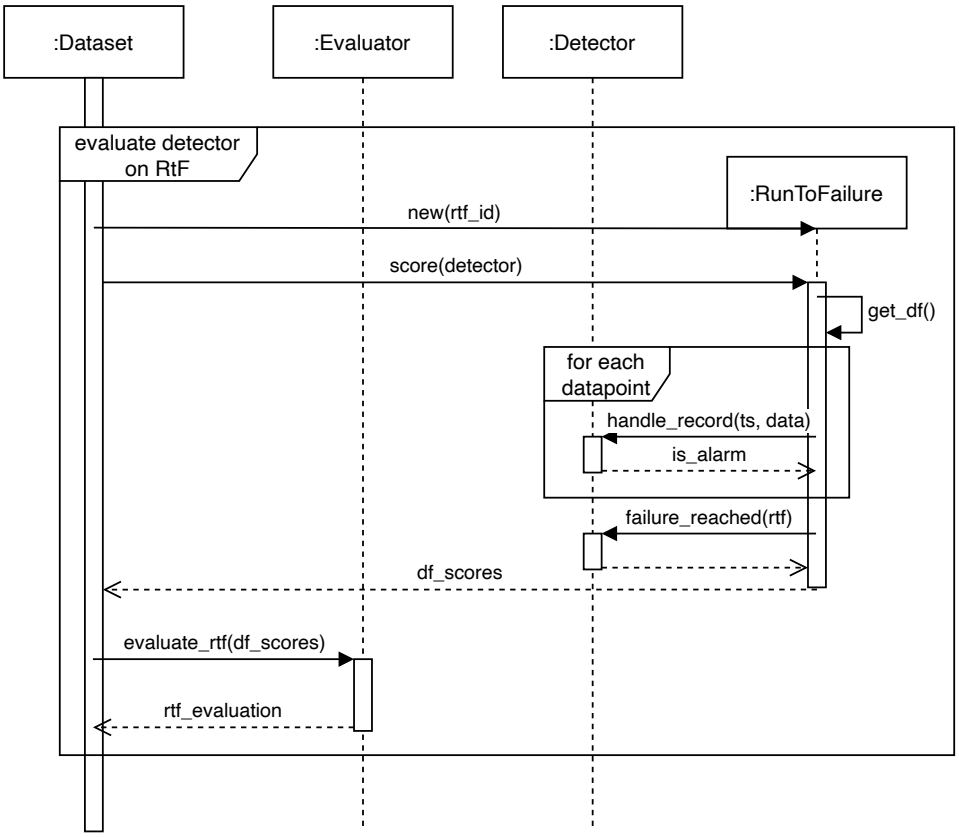



Figure 4.8: Sequence diagram of the test process for a RtF

tifier. Next, the RtF instance loads and iterates over the time series data. Within each iteration, the corresponding data point is passed to the detector, which returns a Boolean value indicating whether an alarm should be triggered or not. After all data points have been processed, the RtF informs the detector that a failure has been reached. Hence, the result that the RtF returns is a time series of Boolean values that indicates when the detector in question would trigger an alarm. Then the resulting time series is passed to the evaluator, who assesses how well the detector was able to predict and possibly prevent the failure.

In summary, an algorithm is thus tested by applying it to each individual RtF before the results are combined first at dataset level and finally at benchmark level. Although the implementation is more complex, such a design allows a very flexible use of the benchmark, as each level can be used on its own. For example, a researcher can easily evaluate an algorithm on individual datasets, models, equipments or RtFs only. Furthermore, despite its complex implementation, the design of the test process is easy to understand and allows researchers to work with it quickly.

### Additional Functionalities

In addition to the core test process described above, the benchmark offers various additional functionalities, which are briefly described below.

First, each component of the benchmark exposes additional methods that enable the researcher to better understand his or her algorithm. For example, both algorithms and evaluators expose plotting functionalities that a researcher can use to analyse how an algorithm performs on individual RtFs. Naturally, the basic plotting methods can be overridden within the implementation of an algorithm or evaluator to further specialize them for the particular implementation. To support advanced plotting, an algorithm may return values other than the Boolean alarm indicator, which can then be passed to the corresponding plotting method. For example, an unsupervised anomaly detection algorithm might want to add the anomaly scores and thresholds to the plot.

Additionally, besides the flexibility that comes from the modular design, all methods related to the test process allow additional optional parameters that enable the user to adapt all configurations at runtime. For example, a researcher might want to analyse how his or her algorithm performs when the lead time becomes longer. For this purpose, all related functions at all levels allow a configuration object to be passed to the evaluator which overrides the default. Similarly, the default parameters of an algorithm can be overwritten. Furthermore, the corresponding methods allow the user to limit the evaluation to individual data sets, models, and equipments, even if the published scoreboard only shows the scores for the entire benchmark.

Next, inspired by the lack of a parameter optimization framework in NAB, the benchmark offers a wrapper class that allows the user to optimize his or her algorithm either for the entire benchmark, specific models, equipments or even individual RtFs. As a result, the user is able to perform sophisticated parameter optimization with only two lines of code. The optimisation framework uses the Bayesian optimisation method, a tool often used for parameter tuning. Bayesian optimisation takes a set of parameters $X = x_1, \ldots, x_n$ as input, suggests a value

for each and evaluates the suggestion based on the result of a provided evaluation function $f(X)$. This procedure is repeated until a sufficient number of points is obtained. Since the optimisation algorithm builds a probabilistic model, each proposal during the optimisation loop is not based on the last results and the corresponding proposals, but on the probability of all previous proposals combined with the corresponding result of the evaluation function. As a result, the Bayesian optimization reaches a more optimal solution much faster than comparable methods like grid search. [94]

Another important detail of the algorithm framework is that each algorithm holds an instance of the current evaluator. As mentioned above, this can be useful, for example, to calculate the RUL labels for past data in order to train a supervised model. It can however also be used to implement parameter optimization within the algorithm itself. On each failure, the algorithm can create an instance of itself, process stored RtFs with it and evaluate the results using the given evaluator.

In summary, the benchmark can be used not only as a means of sharing and comparing algorithms and data sets in the field of prognostics, but it also provides additional functionality that allows researchers to analyse in depth how their algorithms behave on a variety of data sets.

## 4.2 Content

Of course, a framework without content is useless. Therefore, the benchmark already contains several data sets, algorithms, and a novel evaluation metric. Naturally, all additions are implemented using the benchmark framework and therefore adhere to the specifications made above.

### 4.2.1 Datasets

In total, four data sets were added to the benchmark at publication, each of which is briefly presented below, including its source, application, basic information on its content and the pre-processing applied. Due to the principle of minimal preprocessing defined by the benchmark, a detailed data analysis is waived.

#### Hard Drive

Since 2013, Backblaze, a data storage provider, has been publishing detailed quarterly reports on its data centres together with the underlying data. The data consists of daily snapshots taken from each operational disk in their data centres. In addition to basic drive information such as model and serial number, the snapshots include Self-Monitoring, Analysis and Reporting Technology (SMART) statistics reported

by the drive. SMART is a monitoring system natively included in modern hard drives which primary function is to report a variety of indicators related to drive reliability. All data is collected in CSV files and is available for a free download on their website. [21]

For the benchmark, all data from 2013 to the end of 2019 was downloaded. In total, it has an unzipped raw size of about 43 gigabytes, contains 167,747,609 daily snapshots recorded from 366,483 hard drives of 122 different models, including 11,006 hard drive failures. The failures are annotated by a Boolean value indicating the last day the drive was operational before failing. Besides the date, serial number, model, the capacity of the drive and the failure label, it contains 40-62 SMART statistics. The statistics are available both raw and normalized, but are not labelled overall. Thus, there is no information on what the individual SMART values represent. Furthermore, as it is common with SMART data, it is a very sparsely populated dataset. [48] Nevertheless, its size and the fact that it is real data make it a viable addition to the benchmark, even if it is not directly related to PdM.

Of course it had to be reduced in size for use in the benchmark. Because the raw data is grouped by days, with each CSV file containing all the snapshots of all the operating hard drives for a single day, it had to be transformed and grouped by serial number and model. Due to the sheer size of the raw data, the transformation was performed in several steps. First, all data was read from the CSV files and written to a MongoDB, with each collection of the database containing all data for one model. Then in a second step, all data for each model was read, grouped by serial number, and finally spliced into RtFs. During this process all normalized SMART attributes were also removed due to the principle of minimal pre-processing. Since a hard drive is replaced upon failure, each hard drive only has one failure.

Of the 122 models, 8 were finally selected for the benchmark. The selection was made manually and based on two criteria. First, each RtF in the model must have at least 100 records, or each disk must have a lifetime of at least 100 days. From the remaining models, the 8 were then selected with 5 to 10 RtFs each. The idea was to limit the processing time of the benchmark and make it comparable with the other datasets, each of which has less than 10 models.

The final data set contains 89 hard disk drives, or 89 RtFs, with 50,447 records, each with 29 raw SMART attributes and one attribute for the capacity of the drive. On average, one RtF extends over 568 days. Furthermore, due to the sparsity of SMART data, over half of the records contain null values, with some attributes containing more than 70%. Consequently, an algorithm must deal with a great sparsity when trying to predict failures for this dataset.

### Turbofan Engine

The turbofan engine data set is very well known in the research community and therefore does not require a lengthy introduction. It is a public dataset available in the NASA Prognostics Data Repository and contains simulated run to failures of turbofan jet engines. [85] The degradation simulation was carried out using C-MAPSS, which stands for Commercial Modular Aero-Propulsion System Simulation and is a software tool for the simulation of large commercial turbofan engine data. [2] For the dataset, four different combinations of operational conditions and fault modes were simulated, each for a fleet of engines. Each engine starts with varying degrees of initial wear and manufacturing variations, which are unknown but are considered healthy and deteriorate over time. Here time is described as cycles, with each cycle representing the data for one flight. For each cycle, the current state of the engine is described with sensor data and a further three attributes describe the operational setting.

For each of the four operational conditions and fault modes, the data is further divided into train and test set. In the train set, the run to failure ends with system failure, while the data in the test set ends sometime prior failure. Naturally, the original goal of the data set is to predict the RUL values for the truncated test set as best as possible, that is, the remaining flights before the engine fails.

In the benchmark, only the train set could be used since the run to failures in the test set do not end with failure, which violates the definition of a RtF. Nevertheless, since the data is already structured in run to failures, very little effort was required to bring it in the format enforced by the benchmark. Specifically, each operating condition is parsed as a model, and each engine is an equipment. In addition, the cycles were converted into timestamps, with two cycles being one day apart.

In total, the turbofan engine data set, as included in the benchmark, contains data from four models, two of which cover 100 and two 260 pieces of equipment, for a total of 709. Similar to the hard drive data set, the engines are not repaired and therefore contain only one RtF each. Across all models, the RtFs contain a total of 160,359 data points and 24 attributes, none of which have null values. On average, a RtF extends over 255 days.

### Water Pump

The water pump dataset is accessible on Kaggle and contains sensor data along with discrete status labels of a small water pump from a remote area. [12] Unfortunately there is hardly any information available, except that a failure of the pump leads to serious life-threatening problems for people. In total, the data covers 152 days with data points recorded every minute and contains seven failures.

For each data point, a discrete label describes whether the pump is healthy, broken or recovering. Again, there is no information about what these states, especially recovering, actually mean. Nevertheless, since the dataset appears to contain real sensor data, it is considered a viable addition to the benchmark.

To parse it into the required format, all data not marked as healthy was cut out, so that a RtF is either from the beginning of the dataset or from the last recovery state to the next broken state. Since each datapoint has a real time stamp, no pre-processing was necessary there.

Altogether, the water pump data set, as provided in the benchmark, comprises seven RtFs from one equipment. Across all RtFs, the total number of data points is 152,032 over a period of 115 days, while each RtF extends over 15 days on average. Furthermore, each RtF contains 51 attributes, each with an average proportion of missing values below 1%.

### Production Plant

The production plant dataset is also available on Kaggle and contains data of eight components within composite production lines for non-woven materials. [4] It was initially provided by Reifenhäuser Reicofil GmBH & Co. KG for a research project founded by the European Union. In the respective research project, the authors modelled the degradation of one of the components within the line by training a self-organizing map of the healthy state of all other components. [101]

The data is available in CSV files, each file containing all data for one run-to-failure experiment. Upon request, the contributors of the dataset explained that there was a brief break during these two experiments, each approximately 100 timesteps long. However, there is no information why. Within each experiment, the components start in a healthy state and degrade continuously over time. Here, time is indicated by step-wise integer values. Each data point contains 25 attributes plus the time step. For three components the status is represented by five attributes each, for the remaining five by two attributes each. For two experiments, the data is split into two CSV files.

Since each component is represented by only two to five attributes and the authors reported good prediction results for only two of the eight components, the entire production line was declared a single equipment for the benchmark. Therefore, the challenge for an algorithm is to detect these two components within the noise of the entire production line to reliably predict the failure.

During pre-processing, the run-to-failure experiments, which were split into two files, were merged. According to the information provided by the contributor of the data set, the cut was filled with 100 null values to reflect the break. In addition, the integer time steps were replaced by time stamps, with two time steps

being five minutes apart.

In total, the production plant data set, as it is contained in the benchmark, consists of one equipment with eight RtFs and a total of 228,624 data points, with each RtF extending over 1 day and 15 hours on average. Each of the 25 attributes has less than 1% missing values.

### 4.2.2 Algorithms

In its first publication, the benchmark contains seven algorithms, three of which are variations of the algorithm presented in chapter 5, two serve as baselines and two as reference implementations for supervised algorithms. Hereinafter, only the two baseline algorithms and the two supervised algorithms are presented, because the unsupervised anomaly detection algorithm is explained in detail in the next chapter.

#### Baseline Algorithms

There are two baseline algorithms available in the benchmark. The first follows the follows the run-to-fail maintenance strategy, not to be confused with the concept of a RtF presented in this thesis, and never triggers an alarm. Therefore, it serves as a minimum reference in the evaluation to see if an algorithm that tries to predict and prevent failure is at least better than doing nothing. This is particularly important for non-industrial applications where a failure can be relatively inexpensive. Furthermore, it can be used to test if there is a point at which PdM is superior to a simple run-to-fail strategy. Inversely, the second baseline algorithm triggers an alarm at each time step and thus provides a reference for over-maintaining.

Apart from serving as a baseline reference in the evaluation of algorithms, both baselines should also be used when implementing evaluators to ensure that simple maintenance strategies do not lead to good results.

#### Supervised Algorithms

As mentioned above, the algorithm framework provided by the benchmark allows the implementation of any type of algorithm. To serve as an example and also to compare the unsupervised algorithm proposed in the next chapter with supervised approaches, both a basic classifier and a basic regressor are implemented using the eXtreme Gradient Boosting (XGBoost) algorithm. Recall that any algorithm must implement a method that receives the current timestamp along with the current data and returns a Boolean indicating whether or not to raise an alarm. Furthermore, an algorithm can implement a method that is called by the benchmark when a failure is reached, thus at the end of each RtF. For a supervised algorithm, this

method can be used to train a model. First, all RtFs already processed are stored in a suitable data structure. Then, at each failure, the model is trained on all data already seen, that is, on all stored RtFs. For a classifier, this means first calculating the RUL values based on the timestamps and then applying binary labels based on the lead time and a prediction horizon. For a regressor, this means calculating the RUL values based on the timestamps. Accordingly, a classifier applies binary labels based on the RUL values. The model is then used to predict either the RUL or a label for each incoming data point. For the classifier, an alarm is returned if the prediction is positive for a predefined number of consecutive iterations. Similarly, an alarm is returned if the RUL value predicted by the regressor is less than the time delta between the start of the prediction horizon and the time of failure for a predefined number of consecutive iterations. Since the corresponding implementation is straightforward, no details are given here. Instead, an interested reader is recommended to study the publicly available code.

Of course, a supervised model cannot trigger an alarm during the processing of the first RtF and might therefore be inferior to an unsupervised model for the first few RtFs. This assumption will be further examined in chapter 6.

### 4.2.3 Evaluating Algorithms in Predictive Maintenance

As mentioned in section 4.1.2, finding a suitable evaluation measure within the framework is challenging. However, this is not due to the unnatural design of the evaluation framework, but rather to the uniqueness of PdM as an AI problem. The framework just enforces an evaluation metric to adhere to the properties of PdM and at least to the humble knowledge of the author, no standard evaluation metric is suitable for the problem at hand. This is only fortified by the fact that most research on PdM develops their own evaluation metric suitable only to the type of algorithm and task at hand.

In the search for a suitable metric, therefore, it is first discussed why standard metrics from similar applications do not work well in PdM and why they not be used in the framework.

#### Why Standard Evaluation Metrics Do Not Work

Obviously there is a multitude of metrics for each type of application. However, since the algorithms output discrete labels, only metrics used in classification tasks are considered. In related work on PdM, classifiers are usually assessed on the basis of a prediction horizon, which determines how far in the future the failure can occur and the prediction is still considered correct. Based on the prediction horizon, each instance can be labelled either positive or negative. Hence, a classi-

fier can easily be evaluated using standard classification measures such as the area under a precision-recall curve. However, within the proposed framework, such an assessment would not adequately reflect the characteristics of PdM. This is because raising a single alarm within the prediction horizon should be evaluated better than raising one at every timestep because multiple alarms would require more effort from the reliability engineer and would probably inflict more cost. Another source of suitable evaluation metrics could be MISL learning. Recall that PdM was defined as a MISL problem in section 2.1.3. In MISL learning, the data is clustered into bags each containing many instances. A bag is labelled positive if it contains at least one positive instance; otherwise negative. [107] In the proposed evaluation framework, this would mean that if a single alarm is raised within the prediction horizon, the failure is prevented. Although the reasoning is sound, such an assessment is too simplistic for two reasons. Firstly, since the number of bags is naturally small due to the limitation to the number of RtFs, an evaluation based only on the number of prevented failures would prevent a detailed comparison between algorithms. Secondly, although the prevention of a failure is the most important aspect of PdM, its application is only meaningful if the costs are low. Since each alarm triggered by the algorithm requires effort and is likely to incur costs, the total number of alarms should be kept low. A MISL-based evaluation would not take this into account.

In summary, the evaluation metrics used in related work are not applicable because they do not fully adhere to the properties of PdM. Instead, they evaluate an algorithm only from the ML perspective, which is often limited to the specific type of algorithm.

#### Design and Formulation

As elaborated above, finding a suitable evaluation metric from a ML perspective does not work. Instead, the proposed metric is derived from PdM itself. Basically, the idea is to leverage the nature of a RtF and combine it with the overall aim of PdM to reduce maintenance costs.

Before the metric itself is presented, it should be recalled that a RtF is defined as a time series ranging from machine start-up to machine failure. Hence, it contains not only the time series data itself, but also the time of failure, the time of start-up and the lifetime, and therefore maps one entire lifecycle of a machine. The metric uses all this information and not just the time of failure, as is the case with evaluation metrics usually used in PdM.

Essentially, the idea is to assess how well an algorithm predicted the failure, compared to how well it could have predicted it, with the measure being the cost that would have been incurred. Consequently, the evaluation has to compute the

actual, minimum and maximum costs. In maintenance operations, costs increase with each maintenance activity and, of course, in the event of machine failure. For the evaluation metric it is assumed that when used in real maintenance operations, a reliability engineer would schedule a maintenance activity for each alarm indicated by an algorithm to check and possibly repair the machine. Each of these maintenance activities causes costs. However, if several alarms are triggered in succession, the engineer would of course not schedule a maintenance activity for each one, but would wait for the activity to be completed. Consequently, an alarm de-duplication window is introduced. After an alarm, all subsequent alarms within the alarm de-duplication window are ignored. Since the idea of the alarm deduplication window is to represent the average time required to perform the maintenance activity, its length is defined as a ratio of the lead time. Thus, the number of maintenance activities is determined based on the number of alarms remaining after pruning by using the alarm de-duplication window.

Next, it has to be determined if the executed maintenance activities would have prevented the failure. For this purpose, the concept of a relevant maintenance window is introduced, which is defined as the time before the lead time in which at least one maintenance activity must be carried out to prevent the failure. Essentially, it defines the true positive window and is therefore very similar to what related work defines as prediction horizon. In order to counteract the ambiguity associated with this, the following argues for the meaningfulness of a relevant maintenance window. First, from a maintenance point of view, it can be argued that the relevant maintenance window is the window in which a reliability engineer would discover the fault during a maintenance visit. This is a common scenario in a time-based preventive maintenance strategy where a machine is regularly inspected while it is still in operation. If the reliability engineer finds a fault during the check, the machine is repaired. [28] Furthermore, from a research perspective, the use of a self-defined window for labelling can be considered a common practice for predicting tasks. Usually, the necessity of such a window is justified by the inherent ambiguity of the task itself. [103]

The above further illustrated in figure 4.9 for a single RtF ranging from $t_0$ to $t_f$. The lead time is represented as $t_l$, $t_{dw}$ indicates the deduplication windows and $t_r$ the maintenance window. Moreover, the yellowish circles are alarms that were processed as maintenance visits, the transparent red ones indicate other alarms that were triggered but ignored, and the green circle represents an alarm that resulted in a maintenance repair that prevented the failure. The last alarm is ignored because it would not be triggered after the machine was repaired.

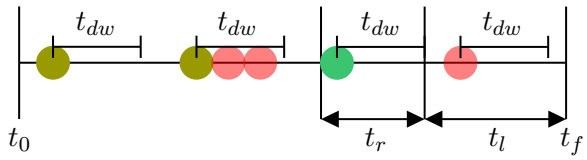



Figure 4.9: Illustration of the cost-based evaluation metric

Using these concepts, the actual costs incurred by an algorithm on a RtF can be simply computed as

$$c_a = n_m * c_m + c_f \tag{4.1}$$

where $n_m \in \mathbb{R}_{\geq 0}$ is the number of maintenance activities that were executed, $c_m \in \mathbb{R}_+$ the cost to perform one activity and $c_f \in \mathbb{R}_+$ the cost of machine failure. Of course, $c_f$ equals zero if the failure was prevented. Since absolute numbers for $c_m$ and $c_f$ are usually not available, especially not for public datasets, a cost rate, denoted as $r_c$, is introduced. The cost rate is defined as

$$r_c = \frac{c_f}{c_m} \tag{4.2}$$

and determines how much more a machine failure costs compared to a maintenance activity. Since only the relative costs are compared in the valuation, the costs of a machine failure can be simplified to

$$c_f = r_c * 1 \tag{4.3}$$

effectively reducing the number of parameters. Hence, the actual relative costs in the illustration above would be

$$c_a = 3 * 1 = 3 \tag{4.4}$$

If the algorithm would not have raised the alarm denoted as a green circle, the costs would be

$$c_a = 3 * 1 + 15 = 18 \tag{4.5}$$

for $r_c = 15$. As the last alarm is now counted, there is still a total of three maintenance activities.

On the basis of the relative actual costs, a comparison with the minimum and maximum costs that could have been incurred makes it possible to estimate how well the algorithm was able to predict the failure.

Naturally, the best possible case is if the algorithm gas triggered only a single maintenance activity and this activity is within the relevant maintenance window. Accordingly, the minimum costs are defined as

$$c_{min} = c_m = 1 \tag{4.6}$$

if the relative cost of one maintenance activity is set to 1 as above.

On the contrary, the worst case scenario is when all possible maintenance activities are triggered, but none in the relevant maintenance window. Hence, the maximum costs are defined as

$$c_{max} = max((n_{fp} + 1) * c_m; n_{fp} * c_m + c_f) \tag{4.7}$$

where $n_{fp}$ is the number of unnecessary maintenance visits. That first term is required because $r_c < 1$ is allowed.

Finally, how well an algorithm performed on a RtF compared to how well it could have performed is calculated as follows:

$$s_{rtf} = \frac{100 - (c_a - c_{min})}{c_{max} - c_{min}} * 100 \tag{4.8}$$

Hence, the maximum score an algorithm can achieve on a RtF is 100, and the minimum score 0.

The idea of relative scores can also be used to aggregate the results first at data set level with

$$s_d = \frac{\sum\limits_{s_{rtf} \in S_{rtf}} s_{rtf}}{100 * |S_{rtf}|} \tag{4.9}$$

and then at benchmark level with

$$s_b = \frac{\sum\limits_{s_d \in S_d} s_d}{100 * |S_d|} \tag{4.10}$$

where $S_{rtf}$ and $S_d$ denote the list of RtF scores and the list of dataset scores respectively. Thus, as with the RtF score, if the proposed evaluation metric is used, the maximum score an algorithm can achieve on a dataset and on the entire benchmark is 100, and the minimum score is 0.

### Implementation

Since the evaluator is fairly easy to implement, details are waived here. However, it should be noted that to simplify the search for suitable parameter settings for a dataset, only the lead time is set as an absolute time delta value. The relevant maintenance window and the alert de-duplication window is set as a fraction of the lead time. Furthermore, $t_l$, $t_r$, and $r_c$ is defined for each dataset, while $t_{dw}$ is defined for the whole benchmark. The reason for this is that the window for deduplication strictly depends on the lead time. How the individual parameters are set for each dataset is shown in table 4.1 and is discussed hereafter.

| Dataset | $t_l$ | $t_r$ | $r_c$ | $t_{dw}$ |
|---|---|---|---|---|
| Hard Drive | 4 days | 20 days | 20 | 3 days |
| Turbofan Engine | 5 days | 15 days | 50.33 | 3 days 18 hours |
| Water Pump | 36 hours | 4 days 12 hours | 7.01 | 27 hours |
| Production Plant | 2 hours | 10 hours | 18.13 | 90 minutes |

Table 4.1: Default parameters of the Maintenance Cost Evaluator

First, the lead time and the relevant maintenance window are solely based on the respective application underlying the datasets and, if available, related work. For the hard drive data set, related work on predicting hard disk failures indicates that the lead time required in a data centre is usually quite short, while the relevant maintenance window or, in the case of classification, the prediction horizon is assumed to be about 10 to 20 days long. [48] Since a RtF extends over an average of almost 570 days, the relevant maintenance window is set at the upper limit of 20 days. For the turbofan engine dataset, although it is undoubtedly the dataset most commonly used in prognostics, it is difficult to find such information. This is probably because the dataset is simulated and exhibits very clear degradation patterns. Furthermore, most research uses regression-based models to predict the RUL and therefore uses regression measures for the evaluation which do not require such parameters. Although some work defines specific evaluation metrics that give greater weight to late predictions than early predictions, their thresholds and ratios are also based on assumptions. In [112], for example, the author's weight predictions 50 cycles, i.e. 50 days before failure, higher, arguing that this is about a quarter of the dataset. A different approach was taken in [74], where the authors use a multiclass classification. Here they assumed a lead time of 10 to 30 days, although their definition of a lead time differs slightly from that assumed in this thesis. Therefore, the lead time for the turbofan engine is set to 5 days and the corresponding maintenance window to 15 days, which takes altogether 20 days. Since there is barely any work on the water pump and the production plant dataset available, their lead times are determined on the basis of personal knowledge of the respective environment. As the data of the first data set originates from a water pump in a remote village, the lead time is set to 36 hours, taking into account the travel time. Furthermore, because a production plant usually is closely monitored, the lead time is set to a quite short interval of only 2 hours. The respective relevant maintenance window sizes are determined based on personal experience and the average RtF length.

Next, since the alert de-duplication is defined as the average time required to perform the maintenance activity, and the lead time as the maximum time, the alert de-duplication window should be a fraction of the lead time. Specifically, it is

estimated that if the machine is always repaired within the lead time, even if the factory is going through a peak period, the average time to complete a repair is about 75%.

As for the cost rates, it can easily be deduced from the definition that they have the greatest influence on the evaluation and should therefore be set with great care. Of course, the best way to find good settings is to know the particular business environment in which the respective machines operate. However, similar to the lead time and the relevant maintenance windows, such knowledge is barely available. Hence, the cost rates are defined based on the performance of the baseline algorithms using the parameters defined above. The idea behind that approach is that both strategies, run-to-fail as well as over-maintaining the machine, are not desirable. For this reason, the cost rate should be set so that the evaluation of both algorithms is minimal. Since the run-to-fail strategy naturally is more effective for lower cost rates and over-maintaining the machine for higher cost rates, the optimal cost rate is at the intersection of both. Figure 4.10 plots the performance of both algorithms on each dataset for different cost rates together with their intersection.

While for the dataset of the water pump, the turbofan engine and the production plant the cost rate is set to the value at the intersection point, the cost rate for the hard drive dataset is set so that the run-to-fail strategy achieves a score of about 88. The reason for this is that in data centres, a hard drive failure usually is cushioned quite well due to storage virtualization techniques. Hence, a run-to-fail strategy should lead to a better evaluation than over-maintaining. For all other datasets, the average rating of both baseline algorithms is 60, which seems reasonable, especially since PdM is considered an economic maintenance strategy for a subset of the machinery only.

Although the above argumentation is partly based on relatively weak assumptions, it can be said to have a more solid foundation than most related work, not least because the design of the evaluation metric is closer to reality.

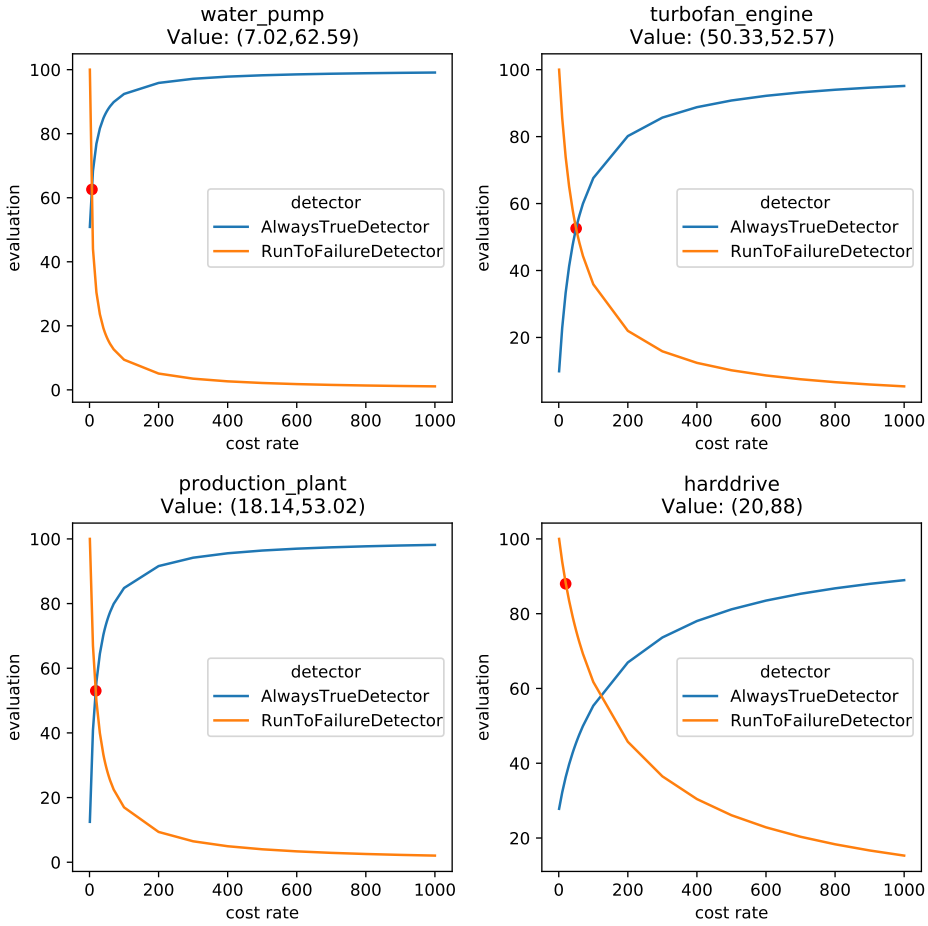



Figure 4.10: Performance of the baseline algorithms for different cost rates and their intersections

### Discussion

To further illustrate how the Maintenance Cost Evaluator works, some practical examples are visualized in figure 4.11. Specifically, the results of applying the algorithm that is proposed in chapter 5 are presented on four RtFs, two from the turbofan dataset and two from the hard drive dataset. There are a few things that stand out and thereby provide valuable insights into how the evaluator works. First, a direct comparison of the figures 4.11c and 4.11d makes it clear that at a low cost rate, the implementation of a run-to-fail maintenance strategy can lead to lower

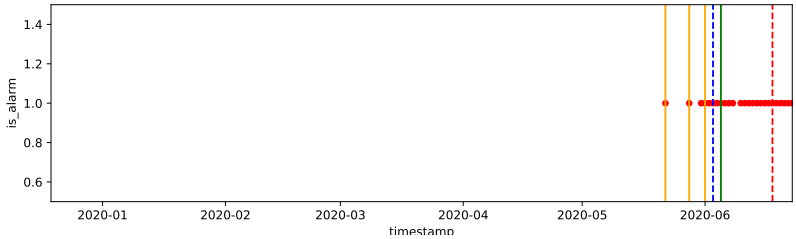
costs than PdM, because any unnecessary maintenance activity leads to relatively high costs compared to a failure. On the contrary, if the cost rate is high, preventing the failure significantly increases the score and is therefore more important than keeping the number of maintenance activities low. Since PdM is the most sophisticated maintenance strategy and is only applied to critical machines, this behaviour reflects reality.

Another property that requires discussion is the decision to evaluate activities within the relevant maintenance window in a discrete manner. Hence, the evaluator does not reflect that triggering an activity close to the begin of the lead time is better than triggering one directly at the beginning of the relevant maintenance window. The reason not to include this is primarily a practical one, because it would require the search for a suitable function and would therefore entail similar challenges as in NAB. Also, since the relevant maintenance window is relatively small, adding such functionality would not significantly change the evaluation results.

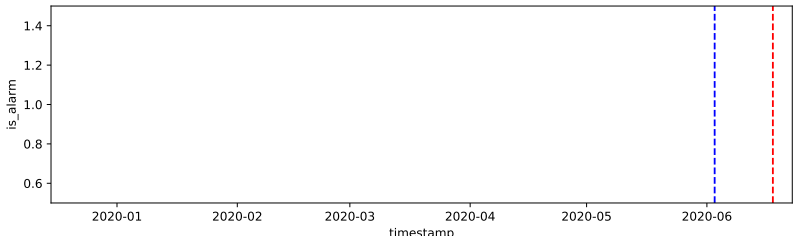
## 4.3 Open Access

Similar to NAB, the benchmark is most valuable as a community tool, benefitting researchers in both academia and industry. Consequently, the source code is completely open and published on GitHub under the permissive GNU Affero General Public License v3.0 (AGPL-3.0 License).[1] Naturally, it follows the standard versioning protocol and uses public issue tracking functionalities. Hence, any changes and additions are openly discussed and communicated. Furthermore, a scoreboard reflects the scores from all contributed algorithms on the corresponding version number using the default parameters for both evaluators and algorithms.

[1] https://github.com/steinroe/prognostics_benchmark

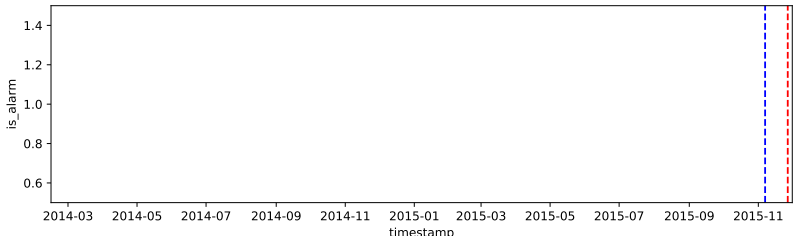



(a) No Failure with $s_{rtf} = 96.8857$ on Turbofan Engine Dataset




(b) Failure with $s_{rtf} = 49.3168$ on Turbofan Engine Dataset


(c) No Failure with $s_{rtf} = 97.3978$ on Hard Drive Dataset




(d) Failure with $s_{rtf} = 91.8103$ on Hard Drive Dataset.

Figure 4.11: Examples of applying the cost-based evaluation metric

— Maintenance Visits — Maintenance Repair

- - Relevant Maintenance Begin - - Lead Time Begin

● Alarms

# Chapter 5

# Leveraging Unsupervised Online Anomaly Detection for Predictive Maintenance

In section 3.1.3 it was pointed out how research in PdM currently focuses on solving very specific problems given a data set and thus lacks generalizability. In addition, most of the algorithms used are supervised learners and therefore require labelled data that is rarely available in practice. Moreover, most models lack interpretability, which prevents a reliability engineer from reproducing the algorithm's decision-making process, thereby limiting the acceptance of PdM within the workforce.

The following chapter proposes an algorithm that leverages the ideas discussed in previous chapters to address the three aforementioned open issues in PdM. First, an overview of the entire model architecture is given before the decisions underlying the model architecture are discussed and each individual element is presented.

## 5.1 Overview

As figure 5.1 illustrates, the model is essentially a set of univariate HTM models whose individual anomaly likelihoods are combined to a single anomaly likelihood that represents the state of the entire system.

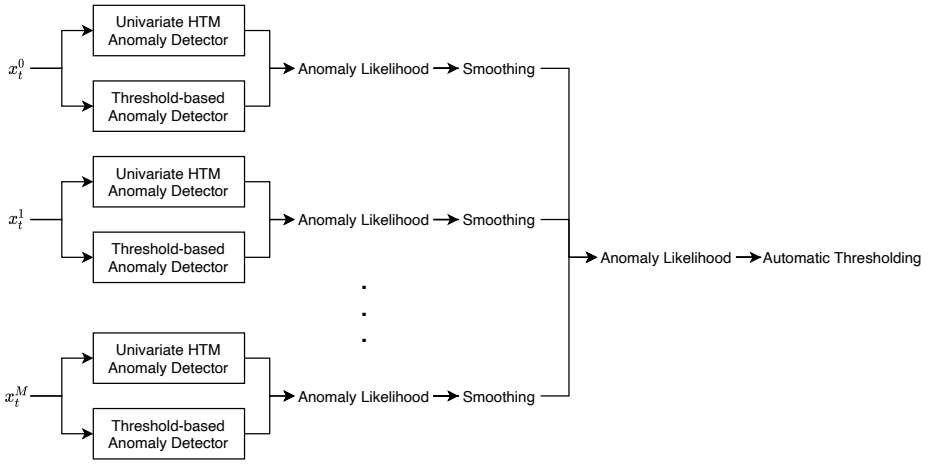



Figure 5.1: Overview of the algorithm

With each incoming multivariate data point, the model first calculates an anomaly likelihood for each individual data stream. The design of the univariate algorithm is very much like the HTM-based anomaly detection algorithm presented in section 3.2.1, but extended by a threshold-based anomaly detector. The idea for the latter comes from the implementation of the NuPIC-based algorithm in the current version of NAB, where the anomaly probability calculated by the HTM-based model is overwritten by the output of the simple threshold-based detector when it detects a spatial anomaly. The respective resulting likelihoods are then smoothed to compensate for delays between anomalies in different data streams before being combined into a single anomaly likelihood that represents the state of the entire system. Finally, an automatic thresholding algorithm calculates an appropriate threshold value that converts the anomaly likelihood into a discrete value indicating whether or not an alarm should be triggered for the current data point.

## 5.2 Why Hierarchical Temporal Memory?

Although it has already been mentioned in the introduction to HTM and its implementation in section 2.3 and section 3.2, it has not yet been explained in depth why a HTM-based model was chosen for univariate online anomaly detection. In the following, the relevant argumentation is presented.

Initially, HTM was chosen since it achieves the best evaluation scores on the NAB. Table 5.1 displays the top scores on NAB after the last commit[1] before start-

[1]Commit 204ccd2174673d2a367cdfadb56320017c5a268f at October 5, 2019

ing this thesis. [11]

| Detector | Standard Profile | Reward Low FP | Reward Low FN |
|---|---|---|---|
| NuPIC | 70.5-69.7 | 62.6-61.7 | 75.2-74.2 |
| CAD-OSE | 69.9 | 67.0 | 73.2 |
| htm.core | 63.1 | 58.8 | 66.2 |
| earthgecko Skyline | 58.2 | 46.2 | 63.9 |

Table 5.1: Top 5 scores on NAB at the time of the start of the thesis. For NuPIC, the range in scores represents runs using different random seeds.

As seen, HTM-based models dominate the scoreboard. However, there are two things that stand out. First, the Contextual Anomaly Detector - Open Source Edition (CAD-OSE) achieves a result close to, and for the reward low false-positive application profile even a better result than NuPIC. Unfortunately, there is no published paper of the algorithm. Furthermore, the author of the algorithm has, upon request, only provided an unofficial, unpublished and difficult to understand description of the algorithm. Consequently, a more in-depth investigation of the CAD-OSE algorithm as an alternative to HTM-based models was waived. Secondly, NuPIC performs significantly better than htm.core, although the latter is actively maintained and contains various bug fixes. Hence, the upcoming sections present an in-depth analysis of both implementations to find and eliminate the shortcomings of htm.core in anomaly detection.

Besides detecting anomalies, related work has also shown that HTM-based models are superior in the broader field of online sequence learning with streaming data. In [37], for example, the authors compare a NuPIC-based implementation with algorithms based on neural networks. Specifically, they compared HTM with Long-Short Term Memory (LSTM), a batch learning algorithm, and Extreme Learning Machine (ELM), which can also learn online. The algorithms were tested on several synthetic datasets, each representing a different property of sequence learning. First, all algorithms were tested in a single prediction experiment in which each sequence has only one possible ending. Both HTM and LSTM achieve perfect prediction, although LSTM requires a sufficiently large batch size and HTM converges much faster. ELM never achieves a perfect prediction. In a second experiment, they altered the dataset to test whether and how fast the algorithms would adapt to concept drift. Again, HTM outperforms both LSTM and ELM, even when using a smaller batch size for the LSTM model. In general, the experiment shows that for any batch learning algorithm, there seems to be a trade-off between prediction accuracy and adaption rate. For short windows, LSTM can adapt relatively quickly, but never reaches a perfect prediction, while for a large

batch size it adapts slowly, but eventually reaches a perfect prediction. In a third experiment, the authors altered the dataset so that each sequence has multiple possible endings. This means, for example, that besides a sequence 1-5-3-7, there are other valid sequences such as 1-5-4-6 and 1-5-1-9. Hence, given the higher-order context, the model has to predict multiple options simultaneously. Here too, the HTM model quickly achieves a perfect forecast. LSTM, on the other hand, is only able to achieve a good prediction on less than three possible endings, and this only for a sufficiently large window size, which leads to a large adaptation time.

It can be summarized that although LSTM can also produce good results, HTM has a number of properties that are undeniably very useful for practical applications of sequence learning on streaming data, such as detecting anomalies in PdM. First, a HTM model learns continuously by design and is therefore able to adapt quickly to a concept drift. Furthermore, it does not need to store training data, which effectively reduces the required resources. Next, the usage of SDRs allows a HTM model to operate well under noisy conditions, which is especially useful for PdM. In addition, the emphasis of HTM research is on imitating the brain rather than achieving a specific behaviour, which ultimately leads to robustness and generalization capabilities. As already observed in related work and emphasized in NAB, a HTM achieves comparable results for different problems with the same parameters derived from biological observations of the neocortex. Since a lack of robustness, generalizability and adaptability are major open issues in PdM, HTM promises to be a very viable algorithm.

## 5.3 Multimodel versus Unimodel Multivariate HTM

In the architecture proposed in section 5.1, a HTM-based anomaly detection model is created for each univariate data stream. All individual anomaly likelihoods are then combined into a single anomaly likelihood that represents the entire system. Of course, this architecture, referred to as a HTM, is not the only way to use HTM in a multivariate manner. Another possible architecture is to use a single HTM model and combine all data points during the encoding phase. Both options come with their benefits and drawbacks.

Naturally, a uni-model architecture benefits from the correlations in the data because it can process the current state as a single system. Therefore, in theory, it should surpass a multimodel HTM architecture if only a few attributes are to be handled. However, the more dimensions are crammed into the spatial pooler, the higher the representational load on each cortical column. Consequently, for a highly dynamic and generalizable system as it is required for PdM, a multimodel architecture seems to be a better practice. In addition, a multimodel ar-

chitecture provides a more fine-grained view of the system and thus enables better interpretability. For example, a reliability engineer could look into the anomaly likelihood for each individual attribute to derive further information on the system status by observing simultaneous spikes. Nevertheless, especially since multivariate HTM is still an open topic in research, a practical comparison of both architectures will be conducted in chapter 6.

## 5.4 HTM-based Univariate Anomaly Detection

The univariate anomaly detection algorithm is very similar to that used by Numenta in its NuPIC-based implementation in NAB. It consists of the HTM-based algorithm presented in section 3.2.1 and a simple threshold-based detector. While the former is responsible for the detection of complex anomalies, the latter reliably detects simple spatial anomalies. Figure 5.2 illustrates the model. For each incoming data point, both the timestamp and the scalar value are encoded using the DateTime encoder and the RDSE respectively. Both encodings are concatenated and then processed by the HTM algorithms, exactly as described in the NuPIC implementation described in section NuPIC. The result of the HTM-based model is an anomaly likelihood value that describes the current predictability of the time series. In parallel, the raw scalar value is processed by a simple threshold-based anomaly detector that detects spatial anomalies. Both likelihoods are then compared, and the greater one is returned.

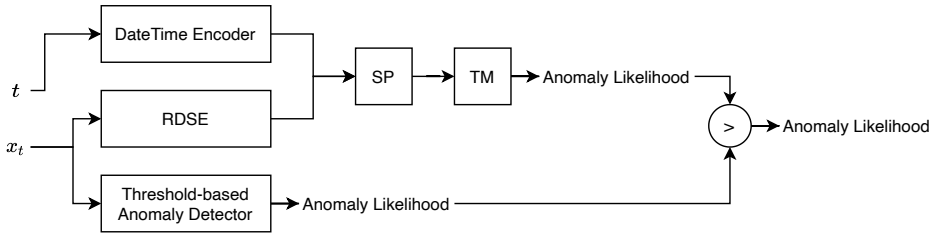



Figure 5.2: Overview of the univariate anomaly detection algorithm

In the following sections, the implementation of both the univariate HTM-based anomaly detection algorithm and the simple threshold-based algorithm is laid out and discussed.

### 5.4.1 htm.core versus NuPIC

In section 5.2 it was shown how the htm.core implementation performs significantly worse than NuPIC on NAB, although actively maintained, has a modern

Python 3.7 implementation and claims to have fixed some bugs. Furthermore, NuPIC only has a Python 2 wrapper and the support for Python 2 was dropped by 01.01.2020. Since there is no other Python implementation of HTM, it was required to find a way of using htm.core without performance loss. For this purpose, it was first analysed how the anomaly detectors implemented in NAB differ. Next, the differences in the APIs of htm.core and NuPIC were examined. Together with a detailed review of the applied bug fixes, it was concluded that the main problem of htm.core is the off-parameter settings. Consequently, a parameter optimization framework for NAB was designed, implemented, and finally contributed to the open-source project. Each of the aforementioned steps is described in the following sections.

### Implementation Differences

In direct comparison, both detectors are implemented very similarly, since they both adhere to the model conceived in [17] and presented in section 3.2.1. Nevertheless, there are a few design decisions and some minor differences that are worth a closer look.

First, both detectors do not use the HTM model as it was presented in section 3.2.1 only but extend it with the simple threshold-based anomaly detector mentioned above. Essentially, it simply checks whether the current value is larger or smaller than all previously seen with regard to a predefined tolerance. Hence, it is designed to capture all spatial anomalies. If a spatial anomaly is detected, it overwrites the anomaly likelihood produced by the HTM model. Section 5.4.3 will provide more details on the implementation of the threshold-based algorithm. Naturally, this raises the question of how much influence the threshold-based anomaly detection extension has on the overall model performance and whether HTM is not able to detect spatial anomalies well. Hence, this will be further investigated in chapter 6. Nevertheless, since both implementations work with the threshold-based anomaly detector, this cannot be the source of their differing performance.

Next, NuPIC is implemented as a research platform for HTM algorithms. Thus, the code itself is rather research-oriented, difficult to understand and does not follow development guidelines. On the other hand, htm.core only intends to provide the core HTM algorithms such as the spatial pooler and the temporal memory along with a Python 3 binding. Therefore, the detector implementation for htm.core is much easier to understand. Nevertheless, after following the implementation of the NuPIC-based detector, no major difference could be detected with respect to the model architecture. Consequently, the difference in the performance must be due to either a faulty implementation or different parameter settings. After the authors

of htm.core reassured[2] that a faulty implementation of the HTM algorithms is most likely not the problem, a detailed comparison of all parameters was required.

### Parameter Differences

Of course, both implementations set the parameters to reflect the architecture of the neocortex and allow generalization. However, there are some minor differences between their respective APIs and parameter settings.

First, as already mentioned in section 3.3.2, in NAB, the algorithms know the minimum and maximum value of each upcoming time series. In the NuPIC-based detector, these values are used to compute the optimal parameters for the RDSE. Specifically, the resolution parameter is set by

$$r = max(0.001, \frac{maxVal - minVal}{numBuckets}) \tag{5.1}$$

where $numBuckets$ is the desired number of buckets set to 130 by default. At least to the modest knowledge of the author, Numenta provided no justification for this setting. Since a RDSE by design is capable of encoding any range of numbers, this approach can be considered slightly biased. In addition, if evaluated strictly, it violates the characteristics of a true online learner as listed in section 2.2.4. The htm.core-based detector does not use this trick, which could affect its performance.

Next, a detailed comparison was made between all parameters of all algorithms involved in the respective HTM anomaly detection models, including their default settings. Since the documentation in NuPIC in particular was confusing and partially contradictory, this undertaking required a detailed analysis of both the Python wrapper and the C++ implementation. The results of the analysis have been contributed to the htm.core repository and can be viewed there.[3] Beside some parameters being off due to non-transparent default settings, the only significant difference between both implementations is that they use different parameters to define the number of active columns within an inhibition area of the spatial pooler. In NuPIC, the number of active columns per inhibition area is set to a fixed value. Recall that the inhibition radius grows with the receptive field of a cortical column. Therefore, by using this parameter, the density of active columns varies. On the contrary, htm.core sets the density of active columns in an inhibition area to a fixed value and also removed the other parameter due to some arbitrary reasons. Although the parameter was reintroduced after some discussions with the author, it did not prove to be beneficial. In fact, setting a fixed value for the number of active columns per inhibition area in an htm.core based model with all other parameters

[2] https://github.com/htm-community/htm.core/issues/792

[3] https://github.com/htm-community/htm.core/pull/794

set to the same values NuPIC uses reduces its NAB score by 1%.[4] Consequently, setting a fixed local inhibition area density is not inherently inferior to setting a fixed number of active columns.

From the above findings, it can be concluded that the inferior performance of htm.core on NAB is due either to issues with its core implementation or to some off-parameter settings, especially for the local area density. Since htm.core runs the same tests NuPIC does, the first can be ruled out. Hence, to improve the performance of htm.core, better parameter settings must be found.

### An Optimization Framework for NAB

Unfortunately, due to its closed architecture, there is no straightforward way to run parameter optimization on NAB. Therefore, in the following, a parameter optimization framework for NAB is presented, which was eventually contributed to the NAB repository.[5]

In designing such a framework, the main task is to provide a way of returning an evaluation given a set of parameters. For NAB, a strict requirement is to not interfere with its implementation. Furthermore, parallel processing must be possible so that any kind of parameter optimization algorithm can work. Recall that NAB works heavily with the filesystem, which makes this a particular challenge. To solve it, the proposed framework is based on Docker and provides each running process with its own virtual file system. First, a Docker image specification, a Dockerfile, is provided in the repository, which installs NAB and, when started, executes the default evaluation process using only the htm.core algorithm. Of course, the same can be done for any algorithm implemented in NAB. Once this image is available, the target function returns an assessment as follows:

1. First, within the target function, a new Docker container is created using the image built beforehand.

2. Next, the currently proposed parameter settings are written to the local file system using the unique container identification for the directory name.

3. To make the parameters available to the detector within the container, the file containing the parameters is copied into the virtual filesystem.

4. Then the container is started, which executes the standard NAB evaluation process inside the container for the given detector, in this case htm.core. During initialization, the detector reads its configuration from the file that was previously copied to the container's virtual file system.

[4] https://github.com/htm-community/NAB/pull/27
[5] https://github.com/htm-community/NAB/pull/31

5. The results of this execution are written to a file inside the container, which is then copied to the host system and read.

6. Finally, the container is destroyed, the corresponding local directory is removed and the previously read NAB score returned.

This design comes with a few benefits. First, it does not make any changes to NAB itself, but rather wraps around it using virtualization techniques. Furthermore, it is very flexible with regard to the parameter optimization algorithm used. The initial contribution contains two examples, one using Bayesian optimization and the other using Particle Swarm optimization. Note that the latter tests multiple parameter settings simultaneously and therefore requires parallel processing. [55] The only disadvantage of this approach is that since NAB does not allow custom parameters on the algorithms, the implementation of an algorithm must be adapted to allow it to be used in parameter optimization. More precisely, instead of using the standard parameters declared within the class itself, the parameters must be read from a file. However, given the closed architecture of NAB, this is a reasonable requirement as it can be implemented for any algorithm with minimal effort and changes.

### 5.4.2 Univariate Anomaly Detection with htm.core

To find well-performing and generalizable parameter settings for htm.core, the parameter optimization framework presented in the previous section was used to optimize the parameters of htm.core on the entire NAB benchmark. The best parameter found after a 12-hour optimization run using the Particle Swarm optimization technique are shown in table B.1 and the respective scores in table 5.2.[6]

| Application Profile | Score Before /w Threshold Detector | Score Before w/o Threshold Detector | Score After /w Threshold Detector | Score After w/o Threshold Detector |
|---|---|---|---|---|
| Standard | 63.08 | 26.78 | 71.30 | 63.1 |
| Reward Low FP | 58.79 | 23.59 | 61.35 | 58.8 |
| Reward Low FN | 66.19 | 28.85 | 76.56 | 66.2 |

Table 5.2: Numenta Anomaly Benchmark scores for htm.core with and without the threshold-based detector before and after the parameter optimization

[6] https://github.com/htm-community/NAB/pull/32

Note that all results shown use the number of active columns per inhibition area parameter instead of the local area density, as the former exceeded the latter by 1%. However, since 1% is not significant, especially because the number of dimensions to be optimized is large, this does not indicate superiority.

Overall, the results show a significant increase in performance through parameter optimization. For the model not using the simple threshold detector also used by NuPIC, the results increased by 130% to 150%. For the model using the threshold detector, the performance increased by 5% to 15%. This result reveals several interesting properties.

First, the threshold-based anomaly detector does have a significant impact on the performance of the HTM algorithm on NAB. Consequently, either a HTM-based anomaly detection model is not able to detect spatial anomalies well or the NAB benchmark mainly consists of spatial anomalies which are easily detectable using simple methods. In chapter 6, although improving HTM further than required for use in PdM is not in scope for this thesis, this hypothesis is further examined.

Moreover, using the same model as NuPIC, so including the threshold-based anomaly detector, the performance of the htm.core based model is slightly better than NuPIC on NAB. Table 5.3 displays the scores after optimization.

| | Standard | Reward Low FP | Reward Low FN |
|---|---|---|---|
| htm.core | 71.3 | 61.35 | 76.56 |
| NuPIC | 70.5 | 62.6 | 75.2 |

Table 5.3: Direct comparison of NAB scores of htm.core and NuPIC after optimization

Consequently, the previously established hypothesis that poor parameter settings are the main cause of inferior performance is confirmed. Note that the official scoreboard lists the ratings for htm.core without using the threshold-based anomaly detector, because the repository authors consider the threshold-based anomaly detector a cheat and do not want to use it for the htm.core implementation. Instead, they want to continue research to become comparable to the NuPIC detector without using the threshold-based detector. However, in the model proposed in this thesis, the threshold-based anomaly detector will be included since the benchmark is not for comparing anomaly detection algorithms but rather all type of algorithms on a specific problem.

Since htm.core is now comparable to NuPIC, it is used to detect univariate anomalies in the model proposed in this thesis. Specifically, the univariate anomaly detection model is designed equally to the model described in section 3.2.1 but with the threshold-based anomaly detector. Furthermore, the parameters that lead to the

best results on NAB are set as default parameters. As this decision has concerns, it needs further discussion. First, the parameter optimization was carried out without any cross-validation and may therefore have overfitted the model. However, since NAB contains many different datasets from various domains, a model that performs well can be considered generalizable. Furthermore, a HTM model is designed to work best when it resembles the characteristics of the neocortex. Consequently, if these parameter settings work well in one application, it can be assumed that they resemble the neocortex well, leading to a generalizable model. Nevertheless, this will be further examined in chapter 6.

As for the implementation, it is based on the implementation provided in NAB, but with a minor adjustment to make it a true online learner. As already mentioned, the algorithms in NAB receive the minimum and maximum value of each upcoming time series. While Numenta leverages that information to set the resolution of the RDSE, the htm.core-based implementation does not use it at all and sets the resolution to a fixed value. For the implementation of the model proposed in this thesis a compromise between both approaches is used. During a probationary period $p_{htm}$ the algorithm first collects all incoming data, which is then used during initialization to determine the minimum and maximum values and to specify the resolution of the RDSE in the same way as Numenta.

### 5.4.3 Threshold-based Spatial Anomaly Detector

The threshold-based detector that is used in the proposed model employs a simple method to detect spatial anomalies. Recall that spatial anomalies are data points that are considered anomalous with respect to the rest of the data. Hence, they can be detected by simply checking whether the current value is significantly larger or smaller than any previously seen, which is exactly what the threshold-based detector does. In addition, a tolerance factor $t_r$ is introduced to reduce the number of false-positives in noisy data sets. Algorithm 1 displays how it is implemented. When a spatial anomaly is detected, the algorithm returns an anomaly likelihood of $1$, thereby overriding the anomaly likelihood calculated by the HTM-based model. If no anomaly is detected, it returns $0$.

## 5.5 Combining Univariate Anomaly Likelihoods

How to combine the individual anomaly probabilities in a multimodel HTM system was originally proposed in the supplementary material of [17]. In the model proposed in this thesis a very similar idea is used with only a minor adjustment to allow a more stable implementation.

**Algorithm 1:** Threshold-based Anomaly Detection

**Parameters:** $t_r$
**Init:** $X_{min}, X_{max}$
**Function** `compute(`$X_i$`)`**:**
  $a_s = 0$
  **if** $X_{min} \neq X_{max}$ **then**
    $t = X_{max} - X_{min} * t_r$
    $t_{upper} = X_{max} + t$
    $t_{lower} = X_{max} - t$
    **if** $X_i > t_{upper}$ *or* $X_i < t_{lower}$ **then**
      $a_s = 1$
    **end**
  **end**
  **if** $X_{max}$ *is* $None$ *or* $X_i > X_{max}$ **then**
    $X_{max} = X_i$
  **end**
  **if** $X_{min}$ *is* $None$ *or* $X_i < X_{min}$ **then**
    $X_{min} = X_i$
  **end**
  **return** $a_s$

A possible solution for combining several probabilities is to estimate the complete joint distribution. However, this is a considerable challenge and highly impractical, especially in highly dynamic scenarios like PdM. Consequently, to simplify this, it is assumed that the individual models are independent. This is of course a very strong assumption, which will have it limits in some scenarios. For example, if a temperature of 30 degrees and a pressure of 5 Pascal is considered normal only if they do not occur together, a multimodel based on such an assumption will probably not be able to detect the corresponding collective anomaly since both univariate streams have a low anomaly likelihood. The consequences of this limitation will be further examined in chapter 6.

Nevertheless, the assumption of independence allows the anomaly likelihood to be calculated as the product of the individual anomaly likelihoods.

$$1 - \prod_{i=0}^{M-1} Q(\frac{\tilde{\mu_t^i} - \mu_t^i}{\sigma_t^i}) \tag{5.2}$$

However, this formula only leads to a high anomaly likelihood if all relevant events occur at the same time. In dynamic real-time systems like PdM this is rather un-

likely. Instead, the problems within a system tend to cascade to other areas over time, which introduces a random delay between the data streams and the respective anomaly likelihoods. Since it is these situations that are very valuable to capture, the authors propose a relatively simple adaption to the above formula by introducing a smooth temporal window to each individual stream of anomaly likelihoods. Such a windowing mechanism allows the model to detect spikes in the system that are close to each other but not exactly coincident. In PdM this means that if, for example, the temperature inside a pump spikes a few minutes before the pressure spikes, the system would be able to detect this because the first spike will continue to have an effect for some time. Hence, the anomaly likelihood of a system is calculated as

$$L_t = 1 - \prod_{i=0}^{M-1} 2(G * Q)(\frac{\tilde{\mu_t^i} - \mu_t^i}{\sigma_t^i}) \tag{5.3}$$

where $G$ is a Gaussian convolution kernel:

$$G(x; \sigma) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{x^2}{2\sigma^2}} \tag{5.4}$$

As before, $L_t$ is an indirect measure reflecting the underlying predictability of the models at each point in time. Figure 5.3 shows the effects of applying a Gaussian kernel to a synthetic data set. It is clear to see how the anomaly at $x = 20$ continues to have an effect long after it occurs. Note that the size of the Gaussian kernel $\sigma$ determines how long the effect lasts.

However, for high-dimensional applications such as PdM, the above equation leads to numerical precision errors because it computes the product of many values between 0.1 and 1. Therefore, in the implementation, the anomaly likelihood is calculated on logarithmic scale and as a summation of log probabilities instead of a product of probabilities:

$$L_t = \sum_{i=0}^{M-1} \log(2(G * Q)(\frac{\tilde{\mu_t^i} - \mu_t^i}{\sigma_t^i})) \tag{5.5}$$

Consequently, $L_t$ is a measure of predictability of the entire system on a logarithmic scale, for which a threshold value is set as described in the following section to raise an alarm.

## 5.6 Extreme Value Theory for Automatic Thresholding

The final step in the model is to convert the anomaly likelihood into a discrete value indicating whether or not to raise an alarm at the given timestep. This is done by

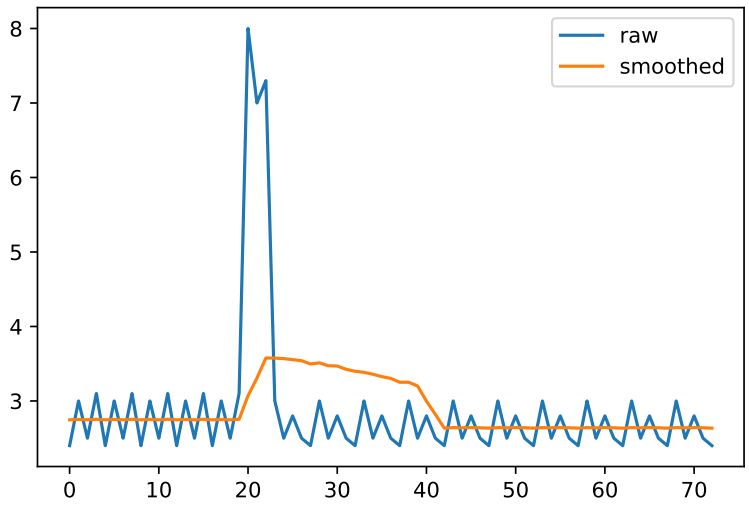



Figure 5.3: Effect of smoothing using a gaussian convolution kernel

automatically setting a threshold on the stream of anomaly likelihoods using the principles of the EVT introduced in section 2.4. In essence, the idea is to combine both the work in [96] and [91] into an algorithm that is capable of automatically setting a threshold on a stream of anomaly likelihoods in an online manner. Although both papers have already been briefly introduced in section 3.2.3, they will be further elaborated in the following to derive the respective adaptation for the application proposed in this thesis.

In [91] the authors propose several algorithms for automatic thresholding, both static and streaming. First, they propose an algorithm for the POT approach which was introduced in section 2.4. Essentially, their algorithm implements the POT method by first finding an initial threshold $t$ on a static data set that is considered high based on the risk factor $q$, retrieving the exceedances above this threshold, and finally fitting a GPD to them using the Grimshaw trick. The result of this algorithm is a threshold value for a static, univariate data set. Next, they propose two algorithms for online anomaly detection on streaming data, Streaming Peaks-Over-Threshold (SPOT) and DSPOT. The SPOT algorithm first executes the POT algorithm on an initial batch of values to get an initial threshold $z_q$. For all subsequent data points, a value $X_i$ that exceeds the current threshold value $z_q$ is marked as an anomaly. Otherwise, if $t < X_i < z_q$, the current threshold value $z_q$ is updated. For all cases $X_i < t$, the value is considered normal and $z_q$ is not updated. Although the SPOT algorithm can be applied in a streaming mode, it assumes that the underlying distribution is stationary. Therefore, the authors propose DSPOT

as an extension that does not require this assumption. Figure 5.4 illustrates how it works.

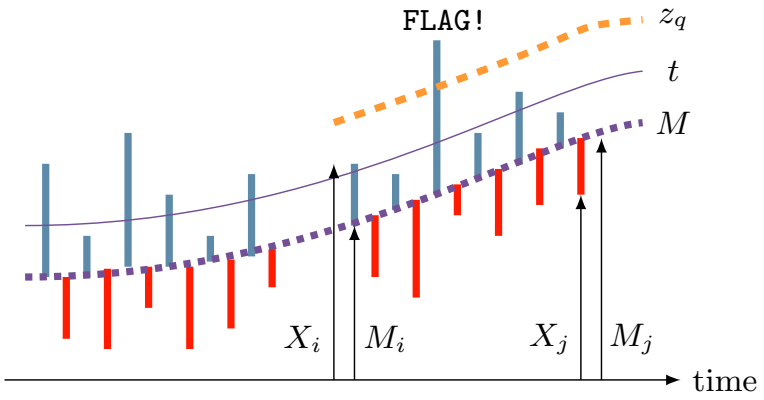



Figure 5.4: Streaming Peaks-Over-Threshold with Drift [91]

Instead of working on the absolute values, the algorithm uses the relative values $X_i^{'} = X_i - M_i$, where $M_i$ is the local behaviour at time $i$. The local behaviour is computed as a moving average

$$M_i = \frac{1}{d} \sum_{k=1}^{d} X_{i-k}^{*} \tag{5.6}$$

where $X_{i-1}^{*}, ...X_{i-d}^{*} = W$ depicts the last $d$ normal observations. Hence, while SPOT assumes stationarity for $X_i$, DSPOT only assumes it for $X_i^{'}$. The DSPOT algorithm works very similar to SPOT, with the only difference that it operates on the local variations instead of the raw values. For more details on the implementation, including possible numerical optimization, the interested reader is referred to [91].

To summarize, with DSPOT, the authors present an algorithm capable of univariate online anomaly detection on non-stationary streaming data leveraging the principles of EVT.

However, the algorithms are applied directly to the raw data and cannot be used on anomaly scores. Specifically, when POT is applied directly to the raw data, the values of interest are at the higher end of the distribution, while when applied to anomaly scores, the anomalies of interest are at the lower end of the distribution.

In [96], the authors build on the idea of automatic thresholding using POT and propose the following adaption in order to capture the anomalies at the lower end

of the distribution. First, they adapt the GPD to

$$\bar{F}_t(x) = \mathbb{P}(t - X > x | X < t) \underset{t \to \tau}{\sim} \left(1 + \frac{\gamma x}{\sigma(t)}\right)^{-\frac{1}{\gamma}} \tag{5.7}$$

where $t - X$ denotes the proportion below a threshold $t$.

In addition, the calculation of the threshold is adjusted as follows:

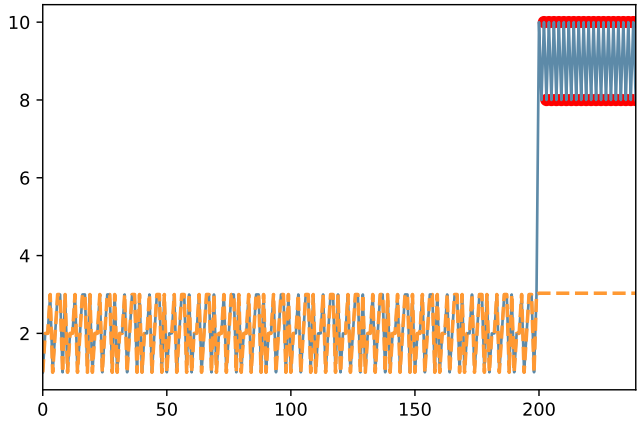
$$z_q \simeq t - \frac{\hat{\gamma}}{\hat{\sigma}} \left( \left( \frac{qn}{N_t} \right)^{-\hat{\gamma}} - 1 \right) \tag{5.8}$$

For better readability the adjusted parts are marked in the formulas. The original formulas can be found in section 2.4.

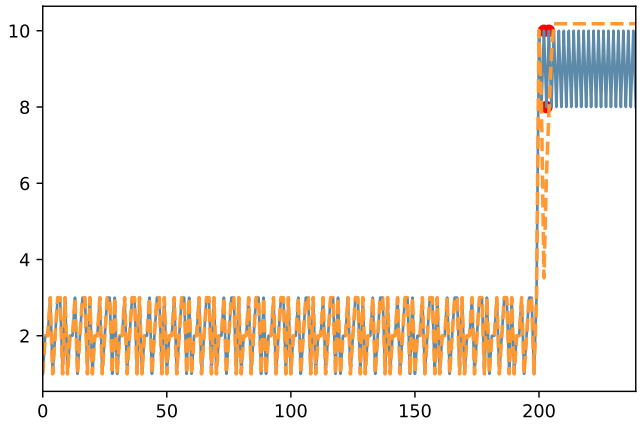
Using these adaptions, the authors successfully leverage the POT algorithm to automatically set a threshold on the anomalies for a static data set. Specifically, in the proposed hybrid learning algorithm, the threshold is set during offline training on a training set.

Consequently, for automatically thresholding anomaly scores in a streaming manner, a combination of both the work in [91] and [96] is required. In concrete terms this means that the adaptions made to the POT algorithm have to be transferred to the DSPOT algorithm to be able to apply it to anomaly scores. Fortunately, the changes required to implement the proposed adjustments are straightforward, requiring only minimal changes to the POT algorithm.

In addition to the modifications suggested in [96], this thesis proposes to make a further modification to the design of the algorithm. In [96], when an alarm is detected on $(X_i - M_i) > z_q$, the threshold value is not updated. While this is reasonable when working with anomaly probabilities on a normal scale, it has its limitations when working in highly dynamic scenarios, especially when working in logarithmic space. Figure 5.5 illustrates the difference between the two methods on a synthetic time series. In 5.5a, the threshold value is updated after the initial peak only when the value falls below 3 again, while in 5.5b the threshold value is quickly adjusted to the new normal. It should be noted that the lines initially overlap only because the threshold value is set equal to the value during the initialization period of the algorithm.




(a) Results without the proposed adaption




(b) Results with the proposed adaption

Figure 5.5: Impact of the proposed adaptation to DSPOT
Value Threshold
Alarms

In addition to adjustments to the design of the algorithm, the implementation provided by [96] has been extensively refactored to fully meet the requirements of a true online algorithm as described in section 2.2.4. Algorithm 2 illustrates the implementation.

**Algorithm 2:** Streaming Peaks-Over-Threshold with Drift for Anomalies

**OnlineDriftSpotForAnomalies**
- **Parameters:** $p_{dspot}, d, q$
- **Init:** $i = -1, A = [], isInitialized = False$
- **Function** `add(`$X_i$`)`**:**
  - $i = i + 1$
  - **if** $i <= p_{dspot} - 1$ **then**
    - $A.push(X_i)$
    - **return** $False, X_i$
  - **end**
  - **if** $isInitialized == False$ **then**
    - **try:**
      - $initialize(A)$
    - **catch:**
      - $A.push(X_i)$
      - **return** $False, X_i$
    - **end**
  - **end**
  - $isAlarm = False$
  - $M_i = W.mean()$
  - **if** $(X_i - M_i) > z_q$ **then**
    - $isAlarm = True$
  - **end**
  - **if** $(X_i - M_i) > t$ **then**
    - $P.push(X_i - M_i - t)$
    - $N_t = N_t + 1$
    - $n = n + 1$
    - $\hat{\gamma}, \hat{\sigma} = grimshaw(P)$
    - $z_q = calcThreshold(q, \hat{\gamma}, \hat{\sigma}, i, N_t, t)$
    - $W = W[1:] \cup X_i$
  - **else**
    - $n = n + 1$
    - $W = W[1:] \cup X_i$
  - **end**
  - **return** $isAlarm, z_q + M_i$
- **end**

**Function** `initialize(`$A$`)`**:**
  $n = |A| - d$
  $M = backMean(A, d)$
  $T = A[d:] - M[:-1]$
  $S = sort(T)$
  $t = S[min(|S| - 3, int(0.98 * n))]$
  $P = T[T > t] - t$
  $N_t = |P|$
  $\hat{\gamma}, \hat{\sigma} = grimshaw(P)$
  $z_q = calcThreshold(q, \hat{\gamma}, \hat{\sigma}, i, N_t, t)$
  $W = A[-d:]$
  $isInitialized = True$
**end**
**end**

During a probationary period $p$, the algorithm just gathers the data and eventually initializes itself using the adapted POT method proposed in [96]. For all subsequent values, the main difference is that in the implementation of [96] and [91] the whole dataset is processed at once in an internal loop, while the algorithm implemented in this thesis exposes a method for processing individual data points one by one. Refactoring was necessary for the algorithm to be included in the benchmark because it enforces stream processing.

Besides the refactoring, the implementation comes with two minor changes to improve generalizability and stability. Instead of setting the initial threshold $t$ to a fixed quantile of $98\%$ as proposed in [91], it is set so that $P$ contains at least two peaks. If there are less than two peaks in $P$ during initialization, the Grimshaw trick can not be applied. Since the quantile of $98\%$ was only empirically determined by the authors, such an adjustment can be justified. However, the initialization still fails if the initial batch of values only contains the same value. To improve stability, initialization is therefore retried for each subsequent data point if it fails. Due to these adaptations, the algorithm can initialize itself even for smaller probationary periods $p < 100$ and thereby offers better generalizability. Otherwise, the algorithm could not be applied to, for example, the hard drive dataset contained in the benchmark because a RtF often contains fewer than $200$ instances.

## 5.7 Parameter Settings

To summarize, the above-presented algorithm is based on an unsupervised multimodel HTM architecture and leverages an adaption of the DSPOT algorithm to automatically set a threshold and convert the anomaly likelihood into a discrete value indicating whether or not to raise an alarm for the current data point. Table 5.4 presents an overview of the tunable parameters of the algorithm along with a short description of each. In total, there are five parameters, of which two control the univariate HTM algorithm and three the DSPOT algorithm. The decision to use relative ratios instead of absolute values was made after first experiments have shown that for all well-performing models $\sigma$ as well as the $p_{dspot}$ are set to a value smaller $p_{htm}$. Since the use of ratios instead of absolute values reduces the space of possible values for the parameters, an optimization algorithm approaches the optimum faster with this change. During initialization, the actual values are computed as

$$\sigma = p_{htm} * r_{\sigma} \tag{5.9}$$

and

$$p_{dspot} = p_{htm} * r_{pdspot} \tag{5.10}$$

respectively.

Since $p_{htm}$ is measured as a number of data points, its value depends heavily on the frequency with which the data was recorded as well as the length of the RtFs. For example, the hard drive dataset is comprised of daily snapshots, where one RtF contains between 100 and 1,000 data points, while the data in the water pump data set is recorded per minute and each RtF holds up to 50,000 data points. Consequently, the number of instances required during initialization naturally differs between these datasets. Therefore, to improve generalizability, each of the parameters listed in table 5.4 is defined by two values, one for a frequency less than one hour and one for a frequency greater than one hour. During initialization, the algorithm automatically detects the frequency based on the first two data points and sets the appropriate parameter set. Hence, the algorithm has a total of 10 parameters that require tuning. The experiments required to confirm the above are conducted in chapter 6.

| Parameter | | Description |
|---|---|---|
| HTM Probationary Period | $p_{htm}$ | The probationary period for all univariate HTM algorithms which is used to set the resolution of the RDSE. |
| Smoothing Kernel Size Ratio | $r_\sigma$ | The size of the Gaussian kernel used for smoothing the univariate anomaly likelihoods defined as a ratio of HTM Probationary Period. |
| DSPOT Probationary Period Ratio | $r_{pdspot}$ | The probationary period of the DSPOT algorithm defined as a ratio of HTM Probationary Period. During that period, the initial batch of data required for the initialization procedure is gathered. |
| DSPOT Depth Ratio | $r_d$ | The length of the rolling window used in DSPOT defined as a ratio of DSPOT Probationary Period. |
| DSPOT Risk Factor | $q$ | Risk defined for the DSPOT algorithm. |

Table 5.4: Parameters of the proposed algorithm

# Chapter 6

# Results & Discussion

This chapter presents the results of all experiments called for in the previous chapters. Besides their presentation, the results are discussed and possible explanations are given for each observed behaviour.

It should be noted that for all experiments carried out on the benchmark presented in chapter 4 a further, non-public dataset has been added. It is referred to as the internal dataset. Unfortunately, not much information can be given, except that it contains data from a machine working in a real environment.

## 6.1 Threshold Detector Experiment on NAB

As shown in section 5.4.2, the performance of the htm.core-based univariate anomaly detector on NAB is improved significantly by incorporating a simple threshold-based detection algorithm into the model. Consequently, either a HTM-based anomaly detection model is not capable of detecting spatial anomalies well, or the NAB benchmark consists mainly of spatial anomalies that are easily detected by simple methods.

To get an idea of how many of the anomalies present in the NAB corpus are spatial anomalies, the threshold-based algorithm is evaluated on NAB without htm.core. The results were contributed to the public GitHub repository[1] and are presented in table 6.1.

[1] https://github.com/htm-community/NAB/pull/41

| Algorithm | Standard | Reward Low FP | Reward Low FN |
|---|---|---|---|
| Threshold Only | 50.83 | 49.94 | 52.56 |
| HTM | 63.1 | 58.8 | 66.2 |
| HTM + Threshold | 71.30 | 61.35 | 76.56 |

Table 6.1: Comparison of HTM and threshold-based algorithm variations on NAB

Looking at these scores, it becomes clear that the simple threshold-based algorithm achieves a comparable result despite its simplicity. It currently ranks 8th out of 13 in the overall scoreboard, surpassing more complex alternatives such as Etsy Skyline and the Twitter anomaly detection algorithm.[2] Consequently, assuming that the spatial anomaly detector captures most of the spatial anomalies, about half of the anomalies contained in NAB are spatial anomalies. Furthermore, a HTM-based algorithm is not able to capture spatial anomalies reliably since adding the threshold-based algorithm increases the score significantly. Of course, these deductions are very basic and require a more detailed analysis of each anomaly and the way each algorithm handles them. Optionally, the anomalies in NAB would be clustered based on the types defined in section 2.2.1, and the performance of each algorithm is compared by type of anomaly. However, since univariate HTM for anomaly detection is not the focus of this thesis, a more in-depth analysis is waived. Instead, the interested reader is referred to [60] and [106].

## 6.2 Verification of Maintenance Cost Evaluator

During its formulation it was claimed that the cost-based evaluation metric introduced in section 4.2.3 is better suited to evaluate algorithms for PdM problems than standard metrics. To verify this assertion, the significance of the evaluations generated by the proposed metric is compared with that of a standard evaluation metric. For this purpose, a window-based evaluation metric based on the F1 score was implemented within the evaluator framework provided by the benchmark.

Hereafter, the F1-based evaluator is briefly discussed before its evaluations are compared with those of the cost-based metric.

### 6.2.1 F1-based Evaluator

Although there are many standard evaluation metrics with which the cost-based metric could be compared, the design of the benchmark and the characteristics of PdM limit the possibilities. Since the algorithms in the benchmark output a boolean

[2] https://github.com/htm-community/NAB accessed at August 19, 2020

value indicating whether or not to raise an alarm, only classification metrics are applicable. As for the required basic truth, the idea of a prediction horizon as introduced in section 2.1.3 is used. Thus all instances within the prediction horizon are labelled as positive and all others as negative, including instances within the lead time. Since the prediction horizon only covers a small part of a RtF, this procedure leads to an imbalanced class distribution. There are multiple standard evaluation metrics suitable for imbalanced classification, of which the F1 score is considered the most reasonable in this case. Instinctively it could also be argued that the F2 result is the better choice for PdM because it rates false negatives more expensive. However, because PdM is classified as a MISL learning problem, the F2 score might lead to misleading evaluations. Furthermore, in related work applying classification to PdM problems, the F1 score is used as well. [72, 25] For the comparison, a F1-based evaluator was implemented in the framework provided by the benchmark. The default parameters for both the lead time and the prediction horizon are set to the same values as the lead time or the relevant maintenance window of the cost-based evaluator. To merge the evaluations at both dataset and benchmark level, the individual evaluations are averaged. As the implementation is straightforward, further details are waived here.

### 6.2.2 Comparative Assessment of Evaluators

Table 6.2 shows the evaluations of all algorithms on the whole benchmark for both metrics.

| Algorithm | Cost-based Evaluation | Window-based Evaluation |
|---|---|---|
| Multimodel HTM | 75.1717 | 0.1687 |
| Run-to-Fail | 59.5312 | 0.0181 |
| Always True | 52.1593 | 0.3601 |
| XGBoost Classifier | 77.0514 | 0.3768 |
| XGBoost Regressor | 76.5526 | 0.3515 |

Table 6.2: Comparison of cost-based and window-based evaluation for different algorithms on the benchmark

It is apparent that the supervised algorithms are rated much better with the window-based evaluation, which is quite reasonable considering the nature of both methods. Since the window-based metric is derived from imbalanced classification problems and thereby requires a positive prediction for every instance within the prediction horizon for a perfect prediction, instead of just one. The classification-based algorithm in particular is therefore evaluated much better since it learns ex-

actly this on the basis of the prediction horizon on previously seen data. Although such an evaluation is useful when comparing supervised algorithms for imbalanced classification problems, it leads to misleading evaluations when applied to PdM, as illustrated in figure 6.1.

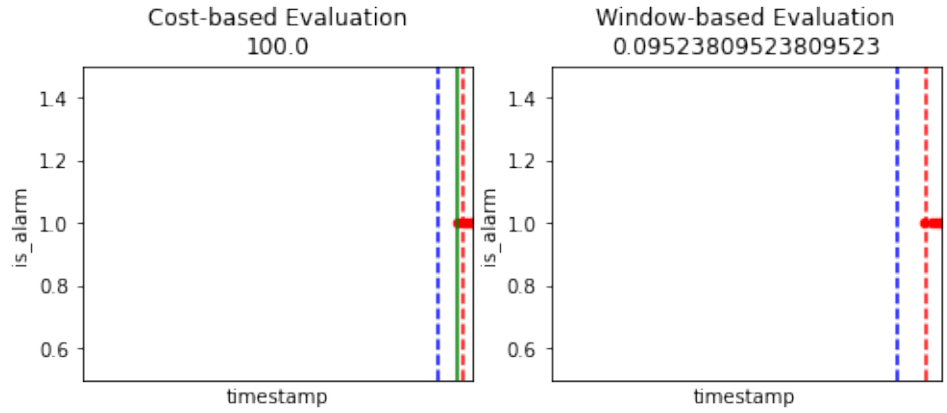



Figure 6.1: Comparison of evaluations of cost-based metric to window-based metric on RtF FD001_10_0 of the turbofan engine dataset

● Alarms — Maintenance Repair
- - Relevant Maintenance / Prediction Horizon Begin - - Lead Time Begin

The algorithm triggers only a single alarm close to the lead time and thus achieves an almost perfect prediction, which is rightly evaluated with a perfect result by the cost-based metric. A window-based metric however evaluates this RtF with a score of 0.09, which is comparable to a 9 of the cost-based metric, since only a fraction of the cases within the prediction horizon were correctly classified as positive. Consequently, the nature of the PdM as a MISL problem limits a generalizable applicability of window-based evaluation metrics, because they prefer supervised algorithms.

Another disadvantage of the window-based metric is that it weights true-positive values much higher than false-positives, resulting in a relatively high score for the Always True baseline algorithm. With the cost-based evaluation metric, the relative weight of both factors can be configured by adjusting the cost rate accordingly, which allows for a much more appropriate evaluation in light of business needs.

It can be concluded that the cost-based metric proposed in this paper is very well suited as a generalisable evaluation metric for PdM problems, as it respects all properties of PdM as an AI problem and allows it to be configured based on business requirements. However, finding reasonable configuration parameters, es-

pecially for the relevant maintenance window, can be challenging.

## 6.3 Comparative Evaluation of Algorithms on the Prognostics Benchmark

The main reason for a benchmark is of course the comparison of algorithms. Table 6.3 shows the results for all algorithms currently contained in the benchmark on the whole corpus using the cost-based evaluation metric with the default configuration. For a recap of the algorithms added to the benchmark the reader is referred to section 4.2.2 and chapter 5. The parameter settings for the multimodel HTM algorithm were obtained by optimization and can be viewed in the appendix in table B.2.

| Algorithm | Evaluation | Max processing time (ms) | Avg processing time (ms) |
|---|---|---|---|
| XGBoost Classifier | 77.0514 | 764.415 | 17.129426 |
| XGBoost Regressor | 76.5526 | 905.744 | 17.389865 |
| Multimodel HTM | 75.1717 | 169225.29 | 96.244571 |
| Run-To-Fail | 59.5312 | 0.657 | 0.001503 |
| Always True | 52.1593 | 0.399 | 0.001512 |

Table 6.3: Evaluation of algorithms on the entire benchmark using the cost-based evaluation metric

Both supervised algorithms are currently on top of the leaderboard. Although the multimodel HTM algorithm is only one point behind, it should be noted that the supervised algorithms are very simple. More sophisticated algorithms would probably perform much better. Nevertheless, the results show that the unsupervised algorithm does have a right to exist, especially since it significantly outperforms both baseline algorithms.

The processing times listed are based on the respective time delta required by the algorithm to process a data point and return whether or not to raise an alarm. Thus the time between two RtFs is not included, following the assumption that there is practically unlimited time between two failures for model training or parameter optimisation. It is noticeable that the unsupervised algorithm needs much more time for processing, which could impair its applicability in highly reactive scenarios. However, the maximum processing time of about 169 seconds only occurs if the algorithm initialises itself after the probationary period. Furthermore, considering that a human being has the feeling of an immediate reaction if the

response time is less than 100 milliseconds, an average processing time of 96 milliseconds is still acceptable. [66]

To get a first impression on the generalisability of the algorithms, table 6.4 displays their evaluations on the individual datasets.

| Algorithm | Hard Drive | Water Pump | Internal | Turbofan Engine | Production Plant |
|---|---|---|---|---|---|
| XGBoost Classifier | 81.3105 | 70.5113 | 70.904 | 97.6755 | 64.8555 |
| XGBoost Regressor | 86.1589 | 59.796 | 66.3621 | 97.7395 | 72.7065 |
| Multimodel HTM | 80.6719 | 73.4685 | 73.7482 | 70.6425 | 77.3275 |
| Run-To-Fail | 88.258 | 51.715 | 52.813 | 52.5717 | 52.2983 |
| Always True | 35.6763 | 64.3532 | 54.5587 | 52.5712 | 53.6373 |

Table 6.4: Evaluation of algorithms on the individual datasets using the cost-based evaluation metric

It is noteworthy that the HTM-based algorithm performs most consistently across the different datasets despite their variety. Moreover, the higher overall evaluation of the supervised algorithms only seems to be due to an almost perfect performance on the turbofan engine data set. For all other datasets, ther performance is either comparable or worse than the HTM-based algorithm. This indicates a better generalizability of the unsupervised algorithm.

Another observation that follows from the results is that the configuration for the hard disk records may be inaccurate, as the evaluation of 88 achieved by the baseline run-to-fail algorithm seems to be very difficult to exceed. It should therefore be reviewed for further research. Based on the results of the other algorithms, a baseline evaluation of 80 seems reasonable. For the upcoming experiments, however, the default configuration will not be changed to ensure consistency.

Although the results presented in this section are already very promising for the HTM-based algorithm, its architecture and properties are further verified hereafter.

## 6.4 Verification of Unsupervised Failure Prediction Algorithm

As discussed in section 3.1.3, practical implementations of PdM are hampered by several open issues. The HTM-based algorithm was designed to tackle some of these open issues, in particular generalizability, interpretability, adaptability and the requirement for poorly available labelled data. Although the results shown in the previous section are already very promising, both its architecture and properties are further verified hereafter.

### 6.4.1 Verification of Architecture

The proposed algorithm is based on two major architectural decisions that need to be verified: The decision for a multimodel architecture as opposed to a unimodel architecture and to apply the principles of the EVT to define a dynamic threshold as opposed to simply defining a fixed threshold. It is to be noted that the use of HTM as the underlying algorithm and the inclusion of the simple threshold-based algorithm is already indirectly verified by the respective performances on NAB. Further verification is therefore omitted.

#### Multimodel versus Unimodel

As already mentioned in section 5.3, instead of processing each data stream through a single HTM model and combining the respective anomaly likelihoods, another option is to use a single HTM model and combine the data streams during encoding. Figure 6.2 illustrates the architecture of such an algorithm. In essence, it is very similar to the univariate HTM model introduced with NuPIC, with the only difference that it concatenates the encoded timestamp with the SDRs of all scalar values of the current input instead of just one.

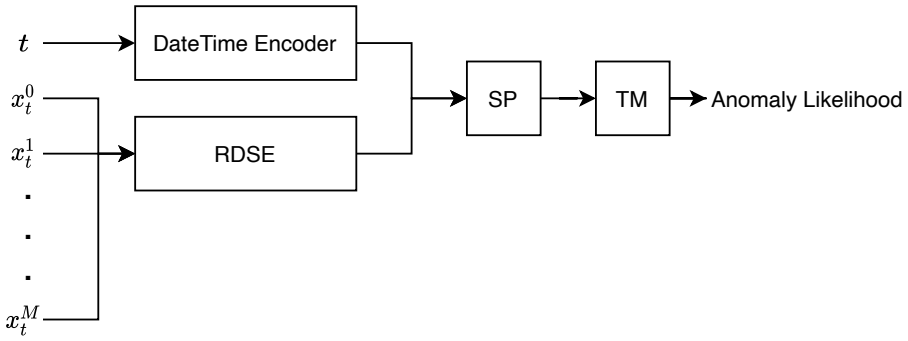



Figure 6.2: Architecture of an unimodel HTM algorithm

At least in theory, an unimodel architecture has its limits if a large number of attributes are to be handled, because the representational load on each cortical column becomes too great. However, at least to the author's modest knowledge, there is no published work that confirms this assumption. The only reference is a discussion in the official HTM forum[3] and an unpublished masters thesis that at least confirmed the superiority of the unimodel architecture for less than six attributes. [26] Due to the weak references, a direct comparison of the two architectures was carried out by comparing their results on the prognostics benchmark using the cost-based evaluation metric. Table 6.5 shows the best results found by parameter optimisation, both overall and for each dataset. The respective parameter settings are displayed in table B.3 and B.2.

| Algorithm | Multimodel HTM | Unimodel HTM |
|---|---|---|
| Evaluation | 75.1717 | 70.0538 |
| Variance | 15.08 | 195.412 |
| Hard Drive | 80.6719 | 88.3937 |
| Water Pump | 73.4685 | 54.7285 |
| Internal | 73.7482 | 57.7187 |
| Turbofan Engine | 70.6425 | 71.9899 |
| Production Plant | 77.3275 | 77.4383 |
| Max processing time (ms) | 169225.29 | 303459.05 |
| Avg processing time (ms) | 96.244571 | 74.3737 |

Table 6.5: Results of unimodel and multimodel HTM algorithm

[3] https://discourse.numenta.org/t/can-htm-be-used-on-multivarient-time-series-problems/3316

Although the overall performance is quite close, on closer inspection the multimodel variant clearly surpasses the unimodel architecture. First, the hard drive dataset is the only dataset on which the unimodel algorithm is clearly superior. However, the achieved score of 88 is the same as the run-to-fail baseline, indicating that the algorithm simply never triggers an alarm. A closer inspection of the results for the individual RtFs confirmed the assumption. For 84 out of 89 RtFs, the unimodel variant never raises an alarm. Consequently, after reviewing the configuration for the evaluation of the hard drive dataset, the performance of the unimodel architecture will decrease significantly, so that the multimodel variant becomes distinctly superior. Furthermore, the variance in the evaluations for the multimodel HTM variant is significantly lower, which indicates a higher degree of generalizability.

Another interesting observation is that the evaluation on the individual datasets seems to be limited by the number of attributes. Table 6.6 shows the average number of attributes for each dataset.

| Dataset | Average Number of Attributes |
|---|---|
| Hard Drive | 20.5 |
| Water Pump | 51 |
| Internal | 33 |
| Turbofan Engine | 24 |
| Production Plant | 25 |

Table 6.6: Average number of attributes for each dataset

The performance of the unimodel approach is significantly worse for the water pump and the internal dataset, which both have more than 25 attributes. Thereby the theoretical limit discussed earlier is confirmed, which makes the multimodel architecture a better practice for a high or unknown number of attributes. Nevertheless, the multimodel architecture as proposed in this thesis merges the individual anomaly probabilities under the principally incorrect assumption of the independence of the individual data streams and thus ignores potentially important correlations. Although this approach seems superior to a uni-model architecture model in practice, further research should be conducted to find an architecture that offers the advantages of both.

### Dynamic versus Static Thresholds

The reason for using the principles of EVT to automatically determine an appropriate threshold, as opposed to defining a static threshold, lies mainly in the promise

of greater generalizability, since no upfront knowledge of the data is required. To verify this assumption, a variation of the algorithm proposed in chapter 5 was implemented in which the DSPOT algorithm is replaced by a static threshold. In order to allow a fair comparison between the two variations, the threshold is defined by two values, one for datasets with a frequency of less than one hour and one for datasets with a frequency of more than one hour. Both threshold values are defined as algorithm parameters and thus benchmark-wide.

Both variants, were optimised on the entire benchmark using the optimisation framework described in section 4.1.3 and the cost-based evaluator as evaluation metric. The best results for each algorithm are presented in table 6.7.

| Algorithm | with DSPOT | with static threshold |
|---|---|---|
| Benchmark | 75.1717 | 64.9134 |
| Variance | 15.08 | 192.5731 |
| Hard Drive | 80.6719 | 87.7274 |
| Production Plant | 77.3275 | 53.6373 |
| Turbofan Engine | 70.6425 | 61.6066 |
| Water Pump | 73.4685 | 67.0368 |
| Internal | 73.7482 | 54.5587 |

Table 6.7: Best evaluations of both variants, dynamic and static threshold, after parameter optimization

The results show that using the DSPOT variant significantly increases overall performance. In addition, since it has a relatively low variance in the individual evaluations, it appears to have greater generalizability. Especially in comparison to the baseline algorithms it becomes evident that DSPOT is a crucial element of the overall architecture, as the static threshold variant hardly exceeds both baselines.

This is fortified in figure 6.3, which displays the sensitivity of the threshold values in the static variant along with the evaluation of the baseline algorithms. Since the hard drive and the turbofan engine dataset both have a frequency greater than one hour and therefore use a different threshold, they are shown in a separate diagram. Given that the final probability of the anomaly is in logarithmic space, the reasonable thresholds are between -200 and 150 instead of between 0 and 1.

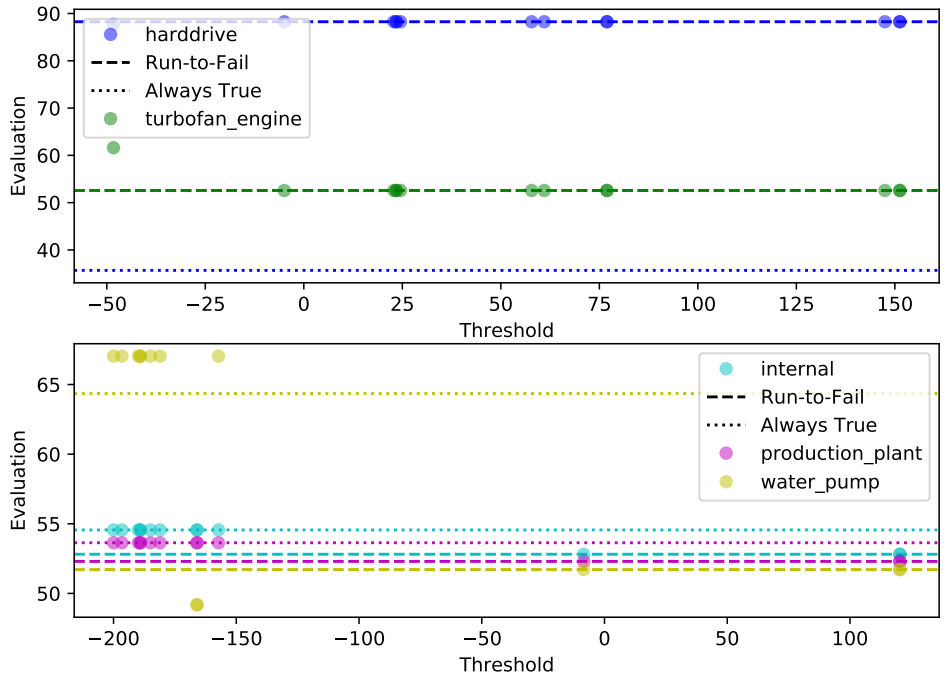



Figure 6.3: Evaluation for each dataset and the benchmark for different static thresholds

The results clearly show that, depending on the threshold, the assessment does not go beyond the baselines in most cases. Moreover, in the few iterations where the algorithm is better compared to the baselines, it is better for only one dataset at a time. One possible explanation for this behaviour is that a static threshold does generalize well, as it can only be well adjusted for a single dataset at a time. This is further fortified by the fact that even after optimisation, the best parameter settings only lead to evaluations that are significantly better than the baselines in two out of five datasets. Consequently, the use of DSPOT instead of a static threshold significantly increases both generalizability and performance.

### 6.4.2 Verification of Properties

#### Unsupervised versus Supervised

The advantage of unsupervised algorithms over supervised ones is that they do not require labelled data. This is particularly useful in PdM, where labelled data is hardly available because machines rarely fail. However, when historical data is

available, supervised algorithms are usually better than unsupervised ones, simply because they use more information. Hence the idea to use the unsupervised algorithm proposed in this thesis in the initial phase of a PdM project until enough historical data is available to train supervised algorithms. Another advantage of unsupervised algorithms is that they do not need to store historical data and therefore require fewer resources. Both statements are tested in the following.

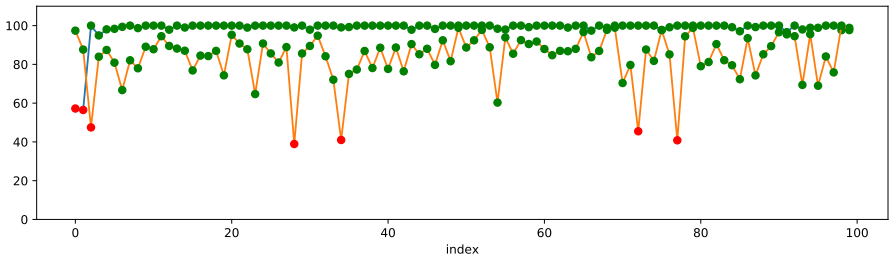
First, the performance of the XGBoost-based classifier implemented in the benchmark is compared with the performance of the HTM-based algorithm at RtF level to see if the performance increases over time. The reason for preferring the classifier over the regressor is that its evaluation is slightly better. Figure 6.4 presents the performance of both algorithms on four different models over time, meaning that the evaluation of each RtFs is shown in the same order in which they are processed by the algorithms. Here some interesting observations can be deduced.

On both turbofan engine models, while the unsupervised algorithm is superior on the first few RtFs, the supervised algorithm shows significantly better performance over time. It is remarkable that after only very few RtFs it almost always perfectly predicts the failure. This can be explained by recalling the nature of the turbofan dataset. Its simulated nature leads to very continuous and predictable degradation patterns, which is why it is also very frequently used for evaluating RUL prediction algorithms. Here, the evaluation metric requires that only a single alarm is triggered within the relevant maintenance window, which is close to the failure. Since supervised algorithms tend to work well near the failure due to the predictability of the dataset, they are evaluated with almost perfect scores with the metric proposed in this thesis. Their good performance close to the failure is clearly reflected in the results of the benchmark and also in related work such as [112]. Interestingly, the unsupervised algorithm is able to predict almost every single failure for the model FD003 but with a lower evaluation. This means that the unsupervised algorithm triggers more alarms overall, but still one within the relevant maintenance window. For model FD004, on the other hand, the unsupervised algorithm performs significantly worse and barely predicts a failure. Again, this can be explained by the nature of the turbofan dataset. Since the HTM-based algorithm is an online learner and adapts to the incoming data stream, there will never be a significant increase in the anomaly likelihood if the continuous deterioration happens too slow, which seems to be the case for FD004. The same conclusion can be drawn from the other two models, with a similar picture for FD001 as for FD003 and a similar picture for FD003 as for FD004.

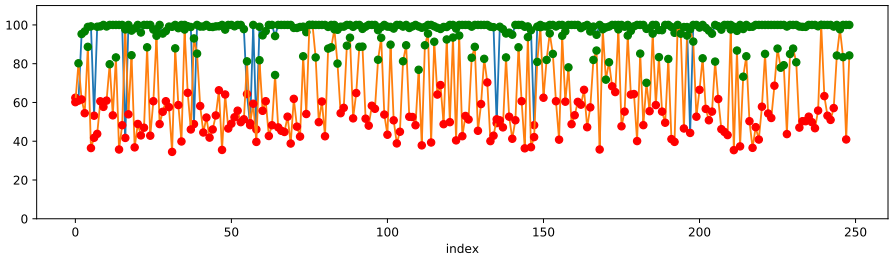
However, a different picture is drawn for the other datasets. In either case, the performance of the unsupervised and the supervised algorithm is very similar, both overall and over time. It should be noted that the cost rate of the hard drive dataset is

set so that the application of a run-to-fail maintenance strategy already achieves an evaluation of 88. On the model presented in figure 6.4c the unsupervised algorithm seems to have a better balance between not predicting the failure and triggering too many alarms. However, as the difference is only marginal, further analysis of the data itself would be necessary to make a clearer statement. For example, since the hard drive dataset is very sparse, important features could not be available, making some failures unpredictable. Another possible explanation is that the supervised algorithms must have already seen data for the same type of failure, and not just any failure, in order to efficiently and reliably predict the same type of failure. The latter is further fortified in figure 6.4d, where the performance of both algorithms on the internal dataset is displayed. Since the data is coming from a single real machine operating in a real environment, the diversity in the type of failures is high. It can be seen that both algorithms predict a large portion of the failures, but the unsupervised algorithm does so at a slightly lower cost. This would also explain the close to perfect evaluation on the turbofan engine dataset. Since the simulation software induced the same type of failure for all engines of the same model, the supervised algorithm is quickly able to predict it very reliably.

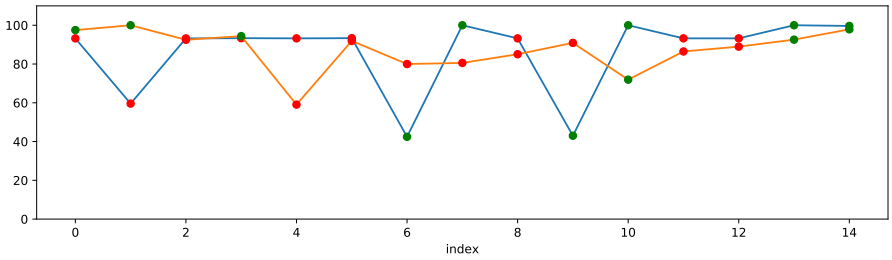
Another interesting observation is that with the predictions of both algorithms in the internal dataset all failures would have been predicted. Although this could be random, it could also mean that unsupervised algorithms are superior for unseen and sudden failures, while supervised algorithms are superior for continuous deterioration and already seen failures.



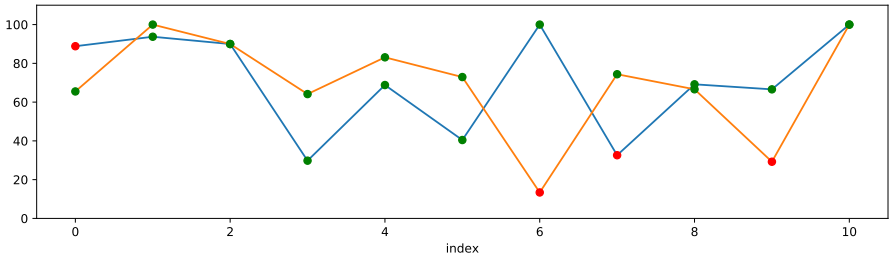

(a) Evaluation on model FD003 of the turbofan engine dataset




(b) Evaluation on model FD004 of the turbofan engine dataset




(c) Evaluation on model WDC WD800AAJS of the hard drive dataset



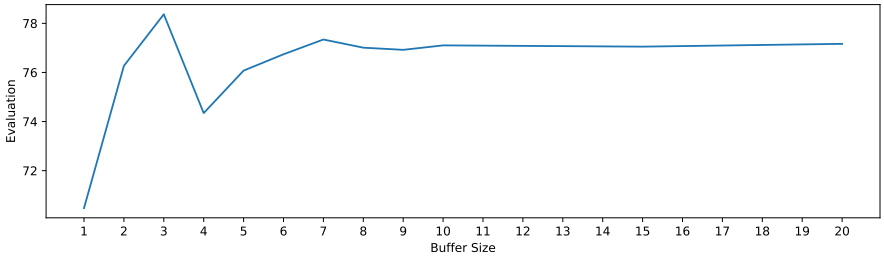

(d) Evaluation on one model of the internal dataset

Figure 6.4: Comparative performance over time of XGBoost-based algorithm and HTM-based algorithm

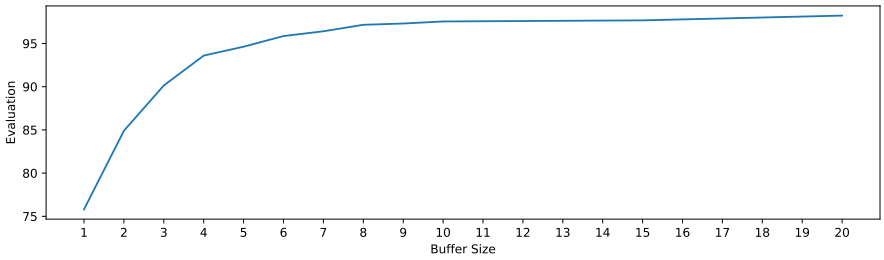
Evaluation Unsupervised — Evaluation Supervised
● Failure Prevented ● Failure not Prevented

In a second experiment, the number of RtFs that the supervised algorithms are allowed to store is restricted. Figure 6.5a shows that for each additional RtF that can be stored, the performance increases significantly until it stabilises at around 6. This behaviour can be observed most clearly for the turbofan engine dataset displayed in figure 6.5b. However, for most other datasets, performance does not increase significantly and stabilises only if the buffer size is greater than the number of RtFs per model. This supports the previous statement that supervised algorithms must see the same type of failure in order to predict it.

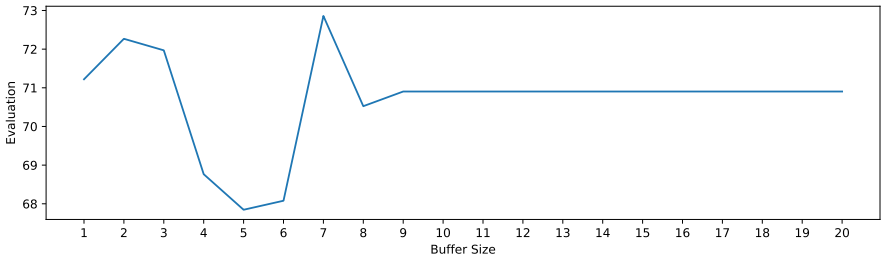
It can be concluded that although supervised algorithms require a certain buffer size, the minimum amount depends strongly on the variety of failure types. A clear recommendation can therefore not be given at this point.




(a) On benchmark




(b) On turbofan engine dataset




(c) On internal dataset

Figure 6.5: Evaluation of XGBoost-based classifier for different buffer sizes

### Generalizability

Since a HTM-based algorithm tries to resemble the function of the neocortex, it should have the same generalizability when the parameters are set according to its biological model. Although the results presented in the previous section already indicate a certain degree of generalizability, it will be further investigated in the following by examining parameter sensitivity.

Of course, a better way to prove generalizability would be to do parameter optimisation using cross-validation. However, due to the size of the benchmark corpus, the duration of a test run is relatively long, which considerably limits the number of runs that can be carried out in a reasonable time. Moreover, similar to NAB, overfitting could be hampered by the wide variety of datasets included in the benchmark alone.

Figure 6.6 shows that the algorithm is quite insensitive to parameter settings. For no dataset, a specific parameter setting is required for the algorithm to achieve a good performance. A direct comparison with figure 6.3 further fortifies that by using DSPOT instead of a static threshold the algorithm becomes much more generalizable. Furthermore, the variance in the evaluation for each datasets is relatively small, which also suggests generalizability, especially in view of the wide variety of datasets.

In summary, the algorithm seems to generalise relatively well or at least not to be sensitive to the parameter settings, especially considering the diversity of datasets included in the benchmark. Nevertheless, this should be further investigated by not optimising for the whole benchmark, but by using cross-validation. Another interesting experiment would be to optimise the algorithm for each dataset to see by how much the performance increases.

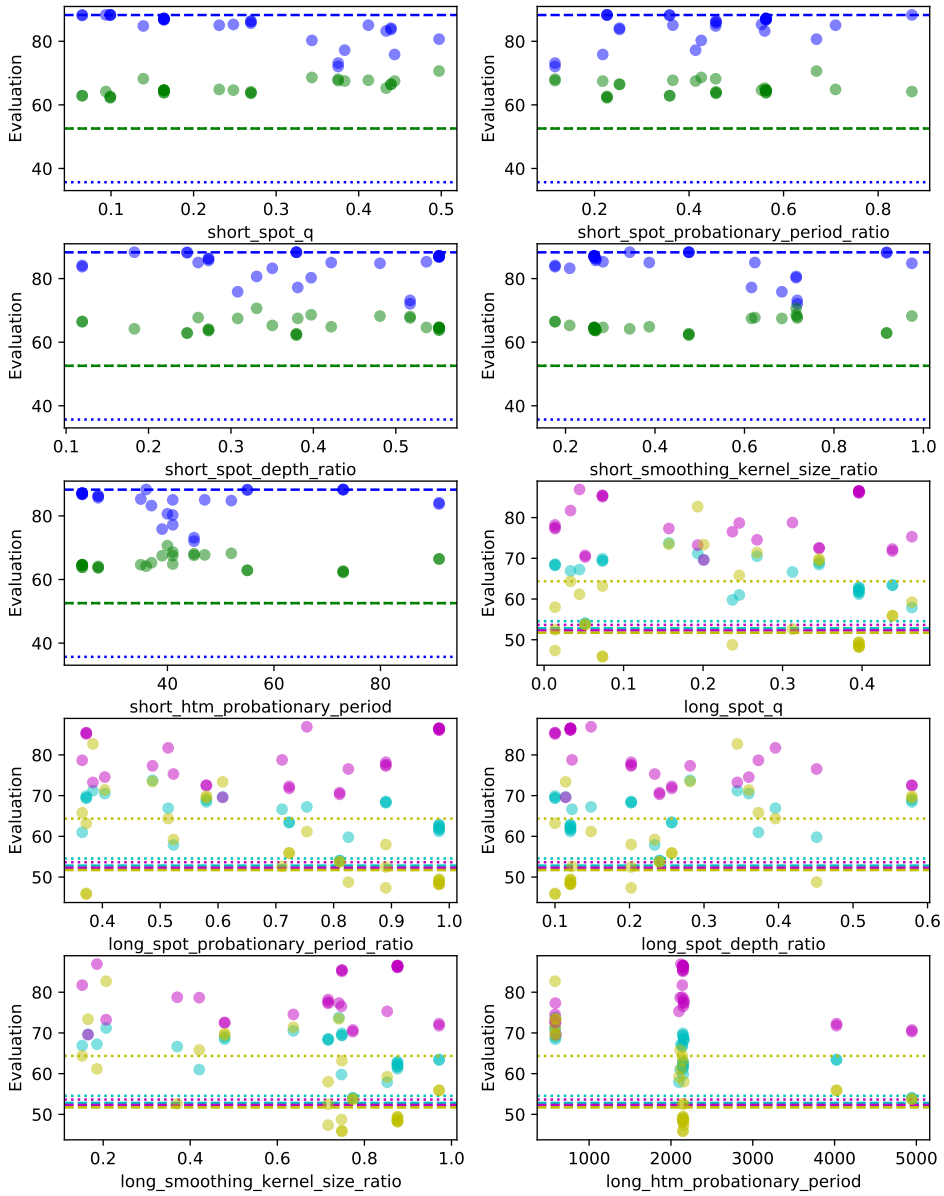



Figure 6.6: Evaluation on each dataset for different parameter settings

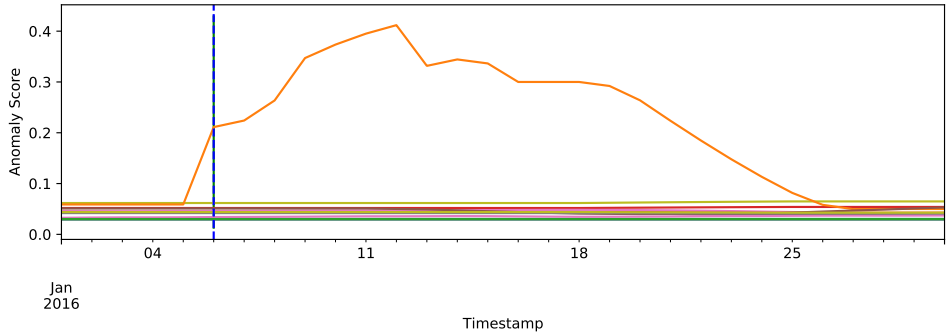
- Run-to-Fail Baseline
- ..... Always True Baseline
- Hard Drive
- Turbofan Engine
- Internal
- Production Plant
- Water Pump

### Interpretability

As discussed in the previous chapter, a multimodel architecture builds a HTM-based anomaly detection model for each individual attribute. Thus, it also produces an anomaly likelihood for each attribute at each time step. This information can be used to provide further insights into the status of individual components or sensors and thereby grants greater interpretability. The figures 6.7 and 6.8 illustrate the interpretability of the multimodel HTM algorithm on one RtF each of the hard drive dataset and the turbofan engine dataset. The respective tables show the four attributes with the highest anomaly likelihood at the time of the alarm.

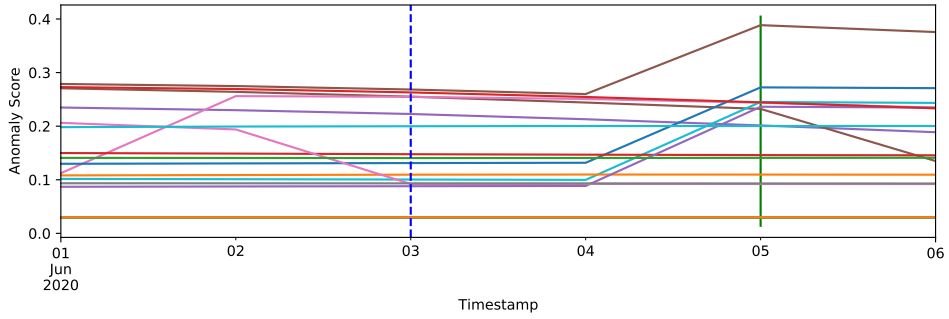
For the RtF of the hard drive dataset it is evident that a significant increase in the anomaly probability of the attribute smart_188_raw has triggered the alarm. In the RtF of the turbofan engine dataset, several values showed increasingly abnormal behaviour from June 4 to June 5, which ultimately triggered the alarm. In both cases, the information provided would help a reliability engineer to analyse the causes. Furthermore, since HTM adheres to the properties of a true online learner, the individual anomaly likelihoods would be displayed in real time in a real-life deployment.

But while this information is very valuable in itself, it would of course be even more useful to show the correlation between the individual attributes and their influence on the overall performance of the machine. However, since a multimodel architecture wrongly assumes that the individual data streams are independent, this would require a different model architecture that is able to detect correlations between the attributes.




| Attribute | Anomaly Likelihood |
|---|---|
| smart_188_raw | 0.2111 |
| smart_12_raw | 0.0616 |
| smart_4_raw | 0.0518 |
| smart_192_raw | 0.0510 |

Figure 6.7: Anomaly likelihoods produced by the univariate HTM models at the time the alarm resulted in a repair for a RtF in the hard drive dataset




| Attribute | Anomaly Likelihood |
|---|---|
| Sensor13 | 0.3884 |
| Sensor8 | 0.2723 |
| Sensor7 | 0.2451 |
| Sensor21 | 0.2444 |

Figure 6.8: Anomaly likelihoods produced by the univariate HTM models at the time the alarm resulted in a repair for a RtF in the turbofan engine dataset

### Adaptability

Another property hoped for by using HTM as the underlying algorithm is the ability to deal with concept drift. Although the adaptability of HTM has already been demonstrated in [37] for univariate sequence prediction, this still needs to be shown for multivariate anomaly detection. Unfortunately, the datasets included in the benchmark do not provide information on whether they are subject to concept drift, nor is it feasible to create a synthetic multivariate dataset with such properties. Instead, concept drift is manually induced into the data for the experiment by splitting individual RtFs. Concretely, the second half of a randomly chosen RtF for which the failure is successfully predicted is concatenated with the first half of all other RtFs of the same model. The assumption is that after the modification all RtFs will have a concept drift at about half of the time series. Finally, the algorithm is applied to the modified RtFs and the resulting evaluation is compared with its evaluation on the original RtFs.

Table 6.8 shows the experiment results for the models FD001 and FD004 of the turbofan engine data set. The reason for choosing these two models is that the algorithm achieves a high score on FD001 and predicts all failures, while on FD004 it scores poorly and predicts only a fraction of the failures. Therefore, a comparison of the evaluations before and after the modifications should give a clear indication of the adaptability.

Rows two to four of the table show the information of the base RtF from which the second half is taken. As mentioned in the experiment setup, the algorithm successfully predicts the failure for both. In rows five to eight the number of predicted failures is compared. On model FD001, the algorithm still predicts all 99 failures, even after the adaption. On model FD004, seven of the previously 38 predicted failures are not predicted anymore after the modification. However, of the 207 failures normally not predicted, 127 are predicted after replacing their second half with a predicted RtF. The algorithm is thus able to adapt to new data and successfully predict the failure, even though the first half of the RtF belongs to another machine for which this was not possible.

Rows nine to 14 show how the average evaluation changes with the modification. In accordance to the number of predicted failures, the average evaluation for the model FD001 does not change, while for model FD004, it increases significantly for normally not predicted failures and decreases slightly for normally predicted failures. Therefore the evaluation changes according to the change in the number of predicted failures, which means that the average number of maintenance activities is stable. This is shown in the rows 15 and 16.

| 0 | Model | FD001 | FD004 |
|---|---|---|---|
| 1 | # of RtFs | 99 | 245 |
| 2 | Base RtF evaluation | 84.5885 | 88.2678 |
| 3 | Base RtF failure prevented | True | True |
| 4 | Base RtF # of maintenance activities | 16 | 15 |
| 5 | # of predicted failures | 99/99 | 38/245 |
| 6 | # of predicted failures adapted RtFs | 99/99 | 158/245 |
| 7 | # of predicted failures of adapted RtFs for normally predicted failures | 99 (-0) | 31 (-7) |
| 8 | # of predicted failures of adapted RtFs for normally not predicted failures | +0 | +127 |
| 9 | Avg evaluation | 84.5542 | 57.1757 |
| 10 | Avg evaluation adapted RtFs | 84.9244 | 79.3614 |
| 11 | Avg evaluation for predicted failures | 84.5542 | 91.8877 |
| 12 | Avg evaluation of adapted RtFs for normally predicted failures | 84.9244 (+0.3702) | 82.7338 (-9.1539) |
| 13 | Avg evaluation for not predicted failures | | 50.8035 |
| 14 | Avg evaluation of adapted RtFs for normally not predicted failures | | 78.7423 (+27.9388) |
| 15 | Avg # of maintenance activities | 17.0606 | 5.6939 |
| 16 | Avg # of maintenance activities adapted RtFs | 16.0505 | 6.5714 |

Table 6.8: Results of the within-model adaptability experiment

Although this already shows that the algorithm must posess a certain degree of adaptability, the results could be biased because as mentioned in section 6.4.2, all failures of a model of the turbofan engine dataset are probably quite similar. Therefore, a second experiment was carried out with a different dataset and across models, which means that the RtF, of which the second half is used, is taken from

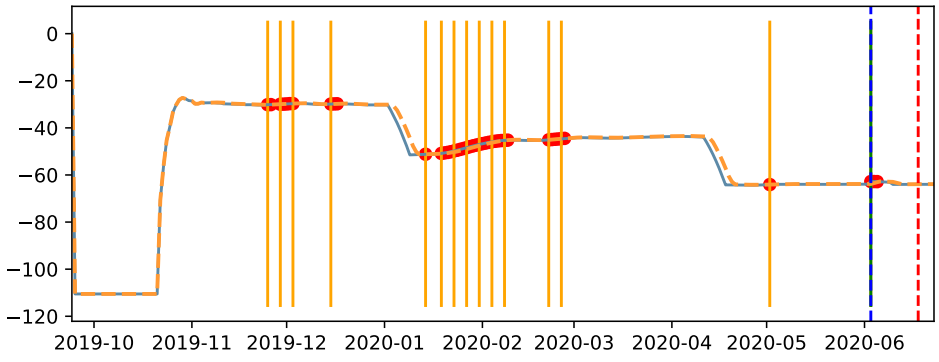
a different model. Table 6.9 shows the results for the model WDC WD20EFRX of the hard drive and FD001 of the turbofan engine dataset using the second half of a RtF of model ST9250315AS and FD004 respectively.

| 0 | Model | WDC WD20EFRX | FD001 |
|---|---|---|---|
| 1 | # of RtFs | 11 | 100 |
| 2 | Base Model | ST9250315AS | FD004 |
| 3 | Base RtF evaluation | 82.1429 | 87.4298 |
| 4 | Base RtF failure prevented | True | True |
| 5 | Base RtF # of maintenance activities | 46 | 16 |
| 6 | # of predicted failures | 7/11 | 99/100 |
| 7 | # of predicted failures adapted RtFs | 10/11 | 87/100 |
| 8 | # of predicted failures of adapted RtFs for normally predicted failures | 7 (-0) | 86 (-13) |
| 9 | # of predicted failures of adapted RtFs for normally not predicted failures | +3 | +1 |
| 10 | Avg evaluation | 84.8108 | 83.7533 |
| 11 | Avg evaluation adapted RtFs | 85.9923 | 84.7331 |
| 12 | Avg evaluation for predicted failures | 92.9462 | 84.2129 |
| 13 | Avg evaluation of adapted RtFs for normally predicted failures | 87.4083 (-5.5379) | 84.6713 (+0.4584) |
| 14 | Avg evaluation for not predicted failures | 70.5738 | 38.2563 |
| 15 | Avg evaluation of adapted RtFs for normally not predicted failures | 83.5144 (+12.9406) | 90.8534 (+52.5971) |
| 16 | Avg # of maintenance activities | 9.7273 | 17.31 |
| 17 | Avg # of maintenance activities adapted RtFs | 25.2727 | 11.1 |

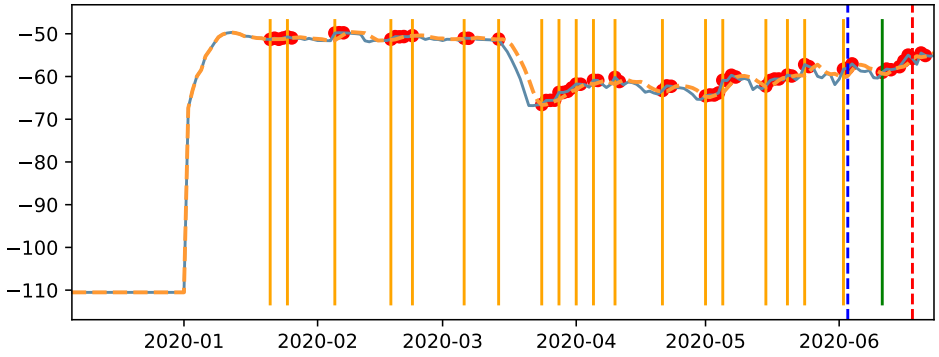
Table 6.9: Results of the across-models adaptability experiment

Remarkably, the results are quite similar to the ones perceived in the previous experiment. For the hard drive dataset, the algorithm is able to increase the number of predicted failures without decreasing the number of normally predicted failures, while the average evaluation increases only slightly. This is due to an increase in the number of maintenance activities. Since the algorithm should output a high anomaly likelihood during adaptation to the concept drift, an increase in maintenance activities is expected. For the turbofan engine dataset, the average evaluation also increases slightly, but instead of an increase in predicted failures, this is due to a decrease in maintenance activities. As illustrated in Figure 6.9, this is because if the algorithm normally triggers many alarms in the second half of the RtFs of the FD001 model, replacement by another RtF for which few activities are triggered in the second half reduces the total number of maintenance activities, despite the concept drift introduced. As can be seen in figure 6.9c, the anomaly likelihood increases sharply at the point where concept drift is induced, but then decreases again quickly. Since in the second half of FD004_4_0, only few false alarms are raised compared to FD001_9_0, the number of alarms is also significantly reduced in the modified RtF.

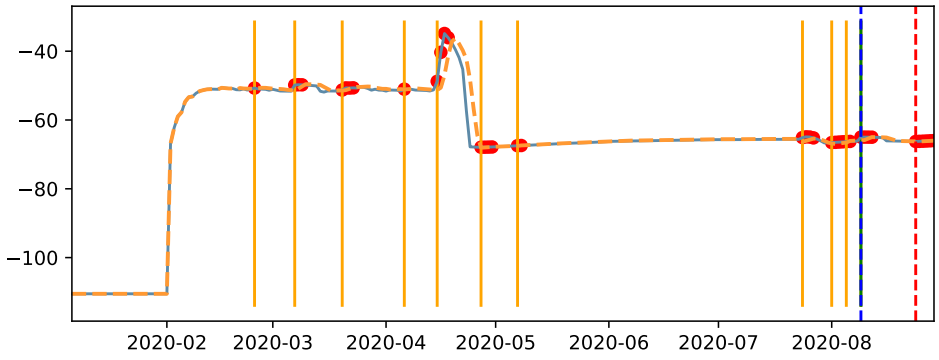
In summary, based on the two experiments performed above, it can be stated that a HTM-based algorithm is capable of adapting to concept drift not only for univariate processing but also for multivariate processing.




(a) $s_{rtf} = 87.4298$ on RtF FD004_4_0




(b) $s_{rtf} = 79.8651$ on RtF FD001_9_0




(c) $s_{rtf} = 91.768$ on adapted RtF where the first half of the time series is taken from FD001_9_0 and the second half from FD004_4_0

Figure 6.9: Performance of HTM-based algorithm before and after the adaption in the across-model experiment

Maintenance Visits
Maintenance Repair
Relevant Maintenance Begin
Lead Time Begin
Anomaly Likelihood
Threshold
Alarms

# Chapter 7

# Conclusion

The following chapter concludes the thesis with a summary of the entire document and a discussion on possible directions for future work.

## 7.1 Thesis Summary

The core of this thesis is to find ways to improve the generalizability of algorithms and the comparability of research in PdM.

Initially, it was briefly discussed how research in PdM is hampered by a large variety of evaluation metrics and limited data availability, which led to the proposal of a benchmark for sharing datasets and comparing algorithms on PdM problems. Furthermore, it was elaborated how the development of generalisable algorithms has the potential to significantly reduce the costs as well as the effort for practical implementations of PdM. As a consequence, it has been suggested that the combination of unsupervised algorithms such as HTM with EVT-based automatic thresholding methods is a promising direction of research towards greater generalizability.

Subsequently, the research area PdM was presented and defined, including a novel definition of PdM as an AI problem that introduces important properties to be considered when working in the field and serves as a basis for the design of the benchmark. Furthermore, important aspects of time-series anomaly detection, HTM theory and EVT were presented, which later act as essential building blocks of the unsupervised failure prediction algorithm.

The provision of background information is followed by an overview of related work on PdM, concluding in an exhaustive list of open issues. Moreover, anomaly detection algorithms and their implementations are presented, including a detailed study of HTM-based variants such as NuPIC. To gain inspiration for the design

of a benchmark, NAB, a benchmark for a univariate streaming anomaly detection algorithm, was discussed and shortcomings and virtues were identified. In addition, the introduction of the NASA Prognostics Data Repository has strengthened the need for a unified way of sharing datasets and comparing algorithms.

Eventually, a novel benchmark for sharing datasets and comparing algorithms on prognostics problems was proposed. Designed based on the definition of PdM as a AI problem, it provides a framework within which researchers can easily contribute datasets and evaluation metrics and implement and compare their algorithms in an open but fair manner. Due to its open architecture and the additional functionalities provided, the benchmark also has the potential to yield major efficiency gains for researchers as a research tool. Upon its first publication, it already contains four open datasets that were gathered, briefly explored and preprocessed in accordance with the specifications provided by the benchmark. Additionally, several algorithms, including baselines, supervised and unsupervised algorithms, are available for reference. To compare the performance of the algorithms on the datasets, a novel cost-based evaluation metric was designed and implemented within the framework provided by the benchmark. The design of the metric was derived from both the business motivation and the AI problem behind PdM, thereby providing a fair and generalizable way to compare any kind of algorithm on PdM problems.

Subsequently to the benchmark, a novel unsupervised failure prediction algorithm was discussed, designed and implemented. The algorithm uses a multimodel architecture and merges the anomaly likelihoods generated by several univariate HTM-based anomaly detection algorithms into a single likelihood, which is then converted with automatic thresholding into a discrete value that indicates whether or not an alarm should be triggered. It is thus able to produce a discrete alarm output without the need for manual interaction, despite being unsupervised. In the process of designing the algorithm, several contributions in the HTM research community have led to a significant performance improvement of a HTM-based univariate anomaly detection implementation and a better understanding of the differences between available open-source implementations of the HTM theory. In addition, several minor adaptations of DSPOT, an EVT-based algorithm for automatic thresholding, were proposed and implemented.

Finally, both the benchmark and the unsupervised failure prediction algorithm were thoroughly studied in a series of experiments. First, a direct comparison of the cost-based evaluation metric proposed as part of the benchmark with a window-based metric often used in PdM has confirmed its validity as a generalisable evaluation metric for PdM problems. Then, a comparative evaluation of the algorithms on the benchmark showed both the credibility of the benchmark and its evaluation metric and the validity of the unsupervised failure prediction algorithm. Next, in

a series of detailed experiments, the main architectural decisions made in the design of the algorithm were confirmed and the hoped-for properties of the algorithm were examined. It was concluded that the algorithm is a suitable substitute for supervised alternatives when little historical data is available, it generalises relatively well, it can adapt to concept drift, and it allows a certain degree of interpretability.

It can be conclusively retained that the rapidly growing interest in PdM from both businesses and researchers assertively confirm the need for better comparability of research and greater generalisability of algorithms, and that the benchmark and the unsupervised failure prediction algorithm represent a significant contribution towards this.

## 7.2 Limitations & Future Work

This section discusses the current limitations of both the benchmark and the unsupervised failure prediction algorithm and suggests some possible ways to overcome them. It also suggests possible extensions and directions for further research.

### 7.2.1 Prognostics Benchmark

Although the benchmark as presented in this paper is already comprehensive, it leaves room for improvement both in the framework itself and the content provided.

A major limitation of the framework is that an algorithm has to be implemented in Python 3 because the benchmark is designed to run in-memory only and does not write intermediate results to the file system as NAB does. This limitation could be lifted by wrapping the relevant modules with a web server, through which an algorithm could receive the data and return its evaluation.

Moreover, while much data has been collected, pre-processed and added to the benchmark, little exploratory analysis has been carried out to better understand the available data and to determine whether or not all failures are predictable. Although such an approach might be considered reasonable as it places the responsibility for data analysis on the researchers who implement the algorithms, there should be at least a minimal analysis of whether each dataset is at all useful for failure prediction. Furthermore, although this work represents an important step towards collecting and unifying all available datasets in PdM, the benchmark still contains too little data. Possible other datasets that can be added are [52] and [14].

Another point of review is that at the moment all pre-processing must be implemented as part of the algorithm. However, if a researcher wants to test different variants but with the same algorithm or the pre-processing becomes more complex, this quickly becomes cumbersome. Therefore, another valuable feature would be

a standardized way to contribute pre-processing functionalities outside the implementation of the algorithm itself. This way, researchers could easily build and test different pre-processing pipelines without having to touch the implementation of the algorithm.

Further research on the evaluation metric could address the search for generalisable ways to derive appropriate parameter settings for the algorithms based on the configuration of the metric. For example, an algorithm could adapt its behaviour based on the cost-rate defined for the dataset being processed.

### 7.2.2 Unsupervised Failure Prediction Algorithm

During the experiments on the algorithm described in section 6.4 several shortcomings were disclosed. These are summarised below and possible directions for further research are suggested.

First, continued research could analyse alternatives to NAB as the underlying anomaly detection algorithm. Particularly promising is the contextual detector that currently leads the scoreboard of NAB. Although the author has not published any paper on it, its source code could be analysed to infer its inner workings. Another direction of research could be to further improve the NAB-based algorithm. In [60] for example, the author proposes a framework that uses high-order prior belief prediction to increase the anomaly detection capabilites of NAB.

Another point of investigation is the multimodel architecture. Although it shows better results in the benchmark than its unimodel alternative, it still assumes independence of univariate data streams and therefore has some undesirable limitations. There are several ways to overcome this. For example, a hybrid approach could be explored that builds unimodel algorithms for batches of data streams, potentially combining the advantages of both the unimodel and the multimodel architecture. Another promising line of research is to find a better way to combine the individual anomaly probabilities that does not require the assumption of independence.

Further significant improvements, in particular regarding the runtime of the algorithm, can be achieved by using an enhanced implementation of the HTM algorithm. In this thesis htm.core was used almost only because it allowed to easily build on the univariate anomaly detection work done by Numenta in [17]. However, other implementations such as Etaler [34] or Brainblocks [42] have features such as GPU support or implement an optimised version of the core algorithms and thus promise a significantly reduced runtime.

# Bibliography


[1] Anomaly Detection with Principal Component Analysis (PCA) - SAP Help Portal. https://help.sap.com/viewer/f50a0b24de8e4968a683e6f926bf1563/1911/en-US/bd98a950503e4277ae9acb3274ddc9a4.html.

[2] Commercial Modular Aero-Propulsion System Simulation (C-MAPSS). NASA.

[3] Numenta — Where Neuroscience Meets Machine Intelligence. https://numenta.com/.

[4] Production Plant Data for Condition Monitoring. https://kaggle.com/inIT-OWL/production-plant-data-for-condition-monitoring.

[5] Prognostics Center of Excellence - Data Repository. https://ti.arc.nasa.gov/tech/dash/groups/pcoe/prognostic-data-repository/.

[6] Stack Overflow Developer Survey 2020. https://insights.stackoverflow.com/survey/2020/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2020.

[7] PHM and Predictive Maintenance. In *From Prognostics and Health Systems Management to Predictive Maintenance 1*, chapter 1, pages 1–13. John Wiley & Sons, Ltd, 2016.

[8] Where are memories stored in the brain? https://qbi.uq.edu.au/brain-basics/memory/where-are-memories-stored, December 2016.

[9] Hierarchical Temporal Memory implementation in Java. Numenta, 2019.

[10] Numenta Platform for Intelligent Computing. Numenta, 2019.

[11] Numenta Anomaly Benchmark, 2020.

[12] "Pump Sensor Data for Predictive Maintenance" on Kaggle, 2020.

[13] Charu C. Aggarwal. *Outlier Analysis*. Springer, 2 edition edition, December 2016.

[14] A. Agogino and K. Goebel. "Milling Data Set", NASA Ames Prognostics Data Repository, 2007.

[15] Subutai Ahmad and Jeff Hawkins. Properties of Sparse Distributed Representations and their Application to Hierarchical Temporal Memory. *arXiv:1503.07469 [cs, q-bio]*, March 2015.

[16] Subutai Ahmad and Jeff Hawkins. How do neurons operate on sparse distributed representations? A mathematical theory of sparsity, neurons and active dendrites. *arXiv:1601.00720 [cs, q-bio]*, May 2016.

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

[18] Wasim Ahmad, Sheraz Ali Khan, M M Manjurul Islam, and Jong-Myon Kim. A reliable technique for remaining useful life estimation of rolling element bearings using dynamic regression models. *Reliability Engineering & System Safety*, 184:67–76, April 2019.

[19] Tarem Ahmed, Mark Coates, and Anukool Lakhina. Multivariate Online Anomaly Detection Using Kernel Recursive Least Squares. In *IEEE INFOCOM 2007 - 26th IEEE International Conference on Computer Communications*, pages 625–633, May 2007.

[20] Nari S Arunraj, Robert Hable, Michael Fernandes, Karl Leidl, and Michael Heigl. Comparison of Supervised, Semi-supervised and Unsupervised Learning Methods in Network Intrusion Detection System (NIDS) Application.

[21] Inc. Backblaze. Backblaze Hard Drive Stats. https://www.backblaze.com/b2/hard-drive-test-data.html.

[22] Radhakisan Baheti and Helen Gill. Cyber-physical systems. *The impact of control technology*, 12(1):161–166, 2011.

[23] Henning Beck, Sofia Anastasiadou, and Christopher Meyer zu Reckendorf. *Das Gehirn*, pages 32–79. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[24] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer (India) Private Limited, 2013.

[25] Francesc Bonada, Lluís Echeverria, Xavier Domingo, and Gabriel Anzaldi. AI for Improving the Overall Equipment Efficiency in Manufacturing Industry. *New Trends in the Use of Artificial Intelligence for the Industry 4.0*, March 2020.

[26] A. Buttignon. *Multivariate Anomaly Detection Using Hierarchical Temporal Memory*. Master, Leuphana Universität, Lüneburg, 2019.

[27] Emmanuel J. Candes, Xiaodong Li, Yi Ma, and John Wright. Robust Principal Component Analysis? *arXiv:0912.3599 [cs, math]*, December 2009.

[28] William W. Cato and R. Keith Mobley. Chapter 2 - Definition of a CMMS. In William W. Cato and R. Keith Mobley, editors, *Computer-Managed Maintenance Systems (Second Edition)*, pages 13–55. Butterworth-Heinemann, Woburn, January 2002.

[29] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM Computing Surveys*, 41(3):1–58, July 2009.

[30] Deepthi Cheboli. *Anomaly Detection of Time Series.* PhD thesis, May 2010.

[31] Kai Chen, Yuhang Cao, Wenwei Zhang, Jiarui Xu, Jiangmiao Pang, Jiaqi Wang, and others. MMDetection: Open MMLab Detection Toolbox and Benchmark. 2019.

[32] Yuanhang Chen, Gaoliang Peng, Zhiyu Zhu, and Sijue Li. A novel deep learning method based on attention mechanism for bearing remaining useful life prediction. *Applied Soft Computing*, 86:105919, January 2020.

[33] 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, September 2018.

[34] An-Pang Clang. Etaler implementation of Hierarchical Temporal Memory, 2020.

[35] Yuwei Cui, Subutai Ahmad, and Jeff Hawkins. Continuous online sequence learning with an unsupervised neural network model. *arXiv:1512.05463 [cs, q-bio]*, April 2016.

[36] Yuwei Cui, Subutai Ahmad, and Jeff Hawkins. The HTM Spatial Pooler—A Neocortical Algorithm for Online Sparse Distributed Coding. *Frontiers in Computational Neuroscience*, 11, 2017.

[37] Yuwei Cui, Chetan Surpur, Subutai Ahmad, and Jeff Hawkins. A comparative study of HTM and other neural network models for online sequence learning with streaming data. In *2016 International Joint Conference on Neural Networks (IJCNN)*, pages 1530–1538, Vancouver, BC, Canada, July 2016. IEEE.

[38] Michael Deighton. *Facility Integrity Management: Effective Principles and Practices for the Oil, Gas and Petrochemical Industries*. Gulf Professional Publishing, February 2016.

[39] Kai Ding, Sheng Ding, Andrey Morozov, Tagir Fabarisov, and Klaus Janschek. On-Line Error Detection and Mitigation for Time-Series Data of Cyber-Physical Systems using Deep Learning Based Methods. In *2019 15th European Dependable Computing Conference (EDCC)*, pages 7–14, September 2019.

[40] N. Duffield, P. Haffner, B. Krishnamurthy, and H. Ringberg. Rule-Based Anomaly Detection on IP Flows. In *IEEE INFOCOM 2009*, pages 424–432, April 2009.

[41] earthgecko. Skyline, 2020.

[42] Jacob Everist and David Di Giorgio. BrainBlocks. The Aerospace Corporation, August 2020.

[43] Fabian Fallas-Moya. *Object Tracking Based on Hierarchical Temporal Memory Classification*. PhD thesis, November 2015.

[44] Sebastian Feldmann, Ralph Buechele, and Vladimir Preveden. Predictive maintenance - from data collection to value creation. Technical report, Roland Berger GmbH, 2018.

[45] Frank E. Grubbs. Procedures for detecting outlying observations in samples. *Technometrics*, 11(1):1–21, 1969.

[46] João Gama, Pedro Medas, Gladys Castillo, and Pedro Rodrigues. Learning with Drift Detection. In *Intelligent Data Analysis*, volume 8, pages 286–295, September 2004.

[47] João Gama, Indrė Žliobaitė, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia. A survey on concept drift adaptation. *ACM Computing Surveys*, 46(4):1–37, April 2014.

[48] Sandipan Ganguly, Ashish Consul, Ali Khan, Brian Bussone, Jacqueline Richards, and Alejandro Miguel. A Practical Approach to Hard Disk Failure Prediction in Cloud Platforms: Big Data Model for Failure Management in Datacenters. In *2016 IEEE Second International Conference on Big Data Computing Service and Applications (BigDataService)*, pages 105–116, March 2016.

[49] Markus Goldstein and Seiichi Uchida. A Comparative Evaluation of Unsupervised Anomaly Detection Algorithms for Multivariate Data. *PLOS ONE*, 11(4):e0152173, April 2016.

[50] J. Hawkins, S. Ahmad, S. Purdy, and A. Lavin. Biological and Machine Intelligence (BAMI). 2016.

[51] Jeff Hawkins and Sandra Blakeslee. *On Intelligence*. Henry Holt and Company, October 2004.

[52] Nikolai Helwig, Eliseo Pignanelli, and Andreas Schutze. Condition monitoring of a complex hydraulic system using multivariate statistics. In *2015 IEEE International Instrumentation and Measurement Technology Conference (I2MTC) Proceedings*, pages 210–215, Pisa, Italy, May 2015. IEEE.

[53] Jerónimo Hernández-González, Iñaki Inza, and Jose A. Lozano. Weak supervision and other non-standard classification problems: A taxonomy. *Pattern Recognition Letters*, 69:49–55, January 2016.

[54] Yuanzhi Huang, Eamonn Ahearne, Szymon Baron, and Andrew Parnell. An Evaluation of Methods for Real-Time Anomaly Detection using Force Measurements from the Turning Process. *arXiv:1812.09178 [cs, stat]*, December 2018.

[55] Lester James V. Miranda. PySwarms: A research toolkit for Particle Swarm Optimization in Python. *The Journal of Open Source Software*, 3(21):433, January 2018.

[56] Veronica Jaramillo Jimenez, Noureddine Bouhmala, and Anne Haugen Gausdal. Developing a predictive maintenance model for vessel machinery. *Journal of Ocean Engineering and Science*, May 2020.

[57] Klemen Kenda, Blaž Kažič, Erik Novak, and Dunja Mladenić. Streaming Data Fusion for the Internet of Things. *Sensors (Basel, Switzerland)*, 19(8), April 2019.

[58] Bartosz Krawczyk, Leandro L. Minku, João Gama, Jerzy Stefanowski, and Michał Woźniak. Ensemble learning for data stream analysis: A survey. *Information Fusion*, 37:132–156, September 2017.

[59] Venu Madhav Kuthadi, Rajalakshmi Selvaraj, and Tshilidzi Marwala. An efficient adaptive preprocessing mechanism for streaming sensor data. In *2015 IEEE 9th International Conference on Intelligent Systems and Control (ISCO)*, pages 1–6, January 2015.

[60] Brody Kutt. *Using High-Order Prior Belief Predictions in Hierarchical Temporal Memory for Streaming Anomaly Detection*. PhD thesis.

[61] Rocco Langone, Carlos Alzate, Bart De Ketelaere, Jonas Vlasselaer, Wannes Meert, and Johan A. K. Suykens. LS-SVM based spectral clustering and regression for predicting maintenance of industrial machines. *Engineering Applications of Artificial Intelligence*, 37:268–278, January 2015.

[62] Alexander Lavin and Subutai Ahmad. Evaluating Real-Time Anomaly Detection Algorithms – The Numenta Anomaly Benchmark. In *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*, pages 38–44, Miami, FL, USA, December 2015. IEEE.

[63] Xiongjun Liu, Ping Song, Cheng Yang, Chuangbo Hao, and Wenjia Peng. Prognostics and Health Management of Bearings Based on Logarithmic Linear Recursive Least-Squares and Recursive Maximum Likelihood Estimation. *IEEE Transactions on Industrial Electronics*, 65(2):1549–1558, February 2018.

[64] Min Luo, Lina Wang, Huanguo Zhang, and Jin Chen. A Research on Intrusion Detection Based on Unsupervised Clustering and Support Vector Machine. In Sihan Qing, Dieter Gollmann, and Jianying Zhou, editors, *Information and Communications Security*, Lecture Notes in Computer Science, pages 325–336, Berlin, Heidelberg, 2003. Springer.

[65] S. Martin-del-Campo and F. Sandin. Online feature learning for condition monitoring of rotating machinery. *Engineering Applications of Artificial Intelligence*, 64:187–196, September 2017.

[66] Robert B. Miller. Response time in man-computer conversational transactions. In *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*, AFIPS '68 (Fall, Part I), pages 267–277, New York, NY, USA, December 1968. Association for Computing Machinery.

[67] Marek Moleda, Alina Momot, and Dariusz Mrozek. Predictive Maintenance of Boiler Feed Water Pumps Using SCADA Data. *Sensors*, 20(2):571, January 2020.

[68] James H. Moor. Turing test. In *Encyclopedia of Computer Science*, pages 1801–1802. John Wiley and Sons Ltd., GBR, January 2003.

[69] V. B. Mountcastle. The columnar organization of the neocortex. *Brain*, 120(4):701–722, April 1997.

[70] Mohsin Munir, Shoaib Ahmed Siddiqui, Muhammad Ali Chattha, Andreas Dengel, and Sheraz Ahmed. FuseAD: Unsupervised Anomaly Detection in Streaming Sensors Data by Fusing Statistical and Deep Learning Models. *Sensors*, 19(11):2451, May 2019.

[71] Mohsin Munir, Shoaib Ahmed Siddiqui, Andreas Dengel, and Sheraz Ahmed. DeepAnT: A Deep Learning Approach for Unsupervised Anomaly Detection in Time Series. *IEEE Access*, 7:1991–2005, 2019.

[72] Athanasios Naskos, Georgia Kougka, Theodoros Toliopoulos, Anastasios Gounaris, Cosmas Vamvalis, and Daniel Caljouw. Event-Based Predictive Maintenance on Top of Sensor Data in a Real Industry 4.0 Case Study. In Peggy Cellier and Kurt Driessens, editors, *Machine Learning and Knowledge Discovery in Databases*, Communications in Computer and Information Science, pages 345–356, Cham, 2020. Springer International Publishing.

[73] Patrick Nectoux, Rafael Gouriveau, Kamal Medjaher, Emmanuel Ramasso, Brigitte Chebel-Morello, Noureddine Zerhouni, and Christophe Varnier. PRONOSTIA : An experimental platform for bearings accelerated degradation tests. In *IEEE International Conference on Prognostics and Health Management, PHM'12.*, volume sur CD ROM, pages 1–8, Denver, Colorado, United States, June 2012. IEEE Catalog Number : CPF12PHM-CDR.

[74] Khanh T. P. Nguyen and Kamal Medjaher. A new dynamic predictive maintenance framework using deep learning for failure prognostics. *Reliability Engineering & System Safety*, 188:251–262, August 2019.

[75] Jack R. Nicholas (Jr.), R. Keith Young, and Elsa K. Anzalone. *Predictive Maintenance Management, 3rd Edition*. Reliabilityweb.com, October 2008.

[76] İzzet Y. öNEL, Engin çAĞLAR, and Ahmet Duyar. New Horizons on Predictive Maintenance. *IFAC Proceedings Volumes*, 42(19):120–125, January 2009.

[77] M. Otahal, D. Keeney, D. McDougall, and others. HTM.core implementation of Hierarchical Temporal Memory, 2020.

[78] Mykoa Pechenizkiy. Predictive analytics on evolving data streams anticipating and adapting to changes in known and unknown contexts. In *2015 International Conference on High Performance Computing Simulation (HPCS)*, pages 658–659, July 2015.

[79] David L. Poole and Alan K. Mackworth. *Artificial Intelligence: Foundations of Computational Agents*. Cambridge University Press, New York, April 2010.

[80] Scott Purdy. Encoding Data for HTM Systems. *arXiv:1602.05925 [cs, q-bio]*, February 2016.

[81] N. Rodríguez-Padial, Marta María Marín, and Rosario Domingo. An Approach to Integrating Tactical Decision-Making in Industrial Maintenance Balance Scorecards Using Principal Components Analysis and Machine Learning. *Complexity*, 2017.

[82] Jose-Raul Ruiz-Sarmiento, Javier Monroy, Francisco-Angel Moreno, Cipriano Galindo, Jose-Maria Bonelo, and Javier Gonzalez-Jimenez. A predictive model for the maintenance of industrial machinery in the context of industry 4.0. *Engineering Applications of Artificial Intelligence*, 87:103289, January 2020.

[83] L. SanjithS and E. George Dharma Prakash Raj. Drift Detection Based Model Selection Framework For Real-Time Anomaly Detection In Iot. *International Journal of Scientific & Technology Research*, 8:646–651, 2019.

[84] SAP. Meßwert- und Zählerstandserfassung - SAP Help Portal. https://help.sap.com/viewer/e72f747389b340229f7fa343975bfa57/1809.001/de-DE/386db65334e6b54ce10000000a174cb4.html.

[85] A. Saxena and K. Goebel. "Turbofan Engine Degradation Simulation Data Set", NASA Ames Prognostics Data Repository, 2008.

[86] Markus Schneider, Wolfgang Ertel, and Fabio Ramos. Expected Similarity Estimation for Large-Scale Batch and Streaming Anomaly Detection. *Machine Learning*, 105(3):305–333, December 2016.

[87] Karlton Sequeira and Mohammed Zaki. ADMIT: Anomaly-based data mining for intrusions. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 386–395, January 2002.

[88] Mahmood Shakir, Frederic Stahl, and Atta Badii. Real-Time Feature Selection Technique with Concept Drift Detection using Adaptive Micro-Clusters for Data Stream Mining. *Knowledge-Based Systems*, August 2018.

[89] Xiao-Sheng Si, Wenbin Wang, Chang-Hua Hu, and Dong-Hua Zhou. Remaining useful life estimation – A review on the statistical data driven approaches. *European Journal of Operational Research*, 213(1):1–14, August 2011.

[90] Christos Siaterlis and Vasilis Maglaris. Towards Multisensor Data Fusion for DoS detection. *Proceedings of the ACM Symposium on Applied Computing*, 1, February 2004.

[91] Alban Siffer, Pierre-Alain Fouque, Alexandre Termier, and Christine Largouet. Anomaly Detection in Streams with Extreme Value Theory. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '17*, pages 1067–1075, Halifax, NS, Canada, 2017. ACM Press.

[92] Nidhi Singh and Craig Olinsky. Demystifying Numenta anomaly benchmark. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 1570–1577, May 2017.

[93] R. K. Sinha, S. K. Sinha, K. B. Dixit, A. K. Chakrabarty, and D. K. Jain. 21 - Plant life management (PLiM) practices for pressurised heavy water nuclear reactors (PHWR). In Philip G. Tipping, editor, *Understanding and Mitigating Ageing in Nuclear Power Plants*, Woodhead Publishing Series in Energy, pages 732–794. Woodhead Publishing, January 2010.

[94] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical Bayesian Optimization of Machine Learning Algorithms. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 2951–2959. Curran Associates, Inc., 2012.

[95] Andrew Spyker, Ruslan Meshenberg, and others. Surus. Netflix, Inc., 2018.

[96] Ya Su, Youjian Zhao, Chenhao Niu, Rong Liu, Wei Sun, and Dan Pei. Robust Anomaly Detection for Multivariate Time Series through Stochastic Recurrent Neural Network. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining - KDD '19*, pages 2828–2837, Anchorage, AK, USA, 2019. ACM Press.

[97] Jianzhong Sun, Fangyuan Wang, and Shungang Ning. Aircraft air conditioning system health state estimation and prediction for predictive maintenance. *Chinese Journal of Aeronautics*, page S1000936119302055, May 2019.

[98] Naoya Takeishi and Takehisa Yairi. Anomaly detection from multivariate time-series with sparse representation. In *2014 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 2651–2656, October 2014.

[99] Kwok L. Tsui, Nan Chen, Qiang Zhou, Yizhen Hai, and Wenbin Wang. Prognostics and Health Management: A Review on Data Driven Approaches. *Mathematical Problems in Engineering*, 2015:1–17, 2015.

[100] Alfonso Velosa, Benoit Lheureux, and W. Roy Schulte. Hype Cycle for the Internet of Things, 2019. Technical report, Gartner.

[101] Alexander von Birgelen, Davide Buratti, Jens Mager, and Oliver Niggemann. Self-Organizing Maps for Anomaly Localization and Predictive Maintenance in Cyber-Physical Production Systems. *Procedia CIRP*, 72:480–485, 2018.

[102] Leandro von Werra, Lewis Tunstall, and Simon Hofer. Unsupervised Anomaly Detection for Seasonal Time Series. In *2019 6th Swiss Conference on Data Science (SDS)*, pages 136–137, June 2019.

[103] Gary M. Weiss and Haym Hirsh. Event Prediction: Learning from Ambiguous Examples. 1998.

[104] Stephan M. Winkler, Michael Affenzeller, Gabriel Kronberger, Michael Kommenda, Bogdan Burlacu, and Stefan Wagner. Sliding Window Symbolic Regression for Detecting Changes of System Dynamics. In Rick Riolo, William P. Worzel, and Mark Kotanchek, editors, *Genetic Programming Theory and Practice XII*, Genetic and Evolutionary Computation, pages 91–107. Springer International Publishing, Cham, 2015.

[105] Haiyue Wu, Aihua Huang, and John Sutherland. Avoiding Environmental Consequences of Equipment Failure via an LSTM-Based Model for Predictive Maintenance. *Procedia Manufacturing*, 43:666–673, January 2020.

[106] Jia Wu, Weiru Zeng, and Fei Yan. Hierarchical Temporal Memory method for time-series-based anomaly detection. *Neurocomputing*, 273:535–546, January 2018.

[107] Ye Xu, Wei Ping, and Andrew T. Campbell. Multi-instance Metric Learning. In *2011 IEEE 11th International Conference on Data Mining*, pages 874–883, December 2011.

[108] Ye Yuan, Guijun Ma, Cheng Cheng, Beitong Zhou, Huan Zhao, Hai-Tao Zhang, and Han Ding. Artificial Intelligent Diagnosis and Monitoring in Manufacturing. *National Science Review*, 7(2):418–429, February 2020.

[109] Jan Zenisek, Florian Holzinger, and Michael Affenzeller. Machine learning based concept drift detection for predictive maintenance. *Computers & Industrial Engineering*, 137:106031, November 2019.

[110] Jianjing Zhang, Peng Wang, Ruqiang Yan, and Robert X. Gao. Long short-term memory for machine remaining life prediction. *Journal of Manufacturing Systems*, 48:78–86, July 2018.

[111] Weiting Zhang, Dong Yang, and Hongchao Wang. Data-Driven Methods for Predictive Maintenance of Industrial Equipment: A Survey. *IEEE Systems Journal*, 13(3):2213–2227, September 2019.

[112] Zeqi Zhao, Bin Liang, Xueqian Wang, and Weining Lu. Remaining Useful Life Prediction of Aircraft Engine based on Degradation Pattern Learning. *Reliability Engineering & System Safety*, 164, March 2017.

[113] Indre Zliobaite and Bogdan Gabrys. Adaptive Preprocessing for Streaming Data. *IEEE Transactions on Knowledge and Data Engineering*, 26(2):309–321, February 2014.

[114] Indrė Žliobaitė, Mykola Pechenizkiy, and João Gama. An Overview of Concept Drift Applications. In Nathalie Japkowicz and Jerzy Stefanowski, editors, *Big Data Analysis: New Algorithms for a New Society*, volume 16, pages 91–114. Springer International Publishing, Cham, 2016.

[115] Kalyani Zope, Kuldeep Singh, Sri Harsha Nistala, Arghya Basak, Pradeep Rathore, and Venkataramana Runkana. Anomaly Detection and Diagnosis

in Manufacturing Systems: A Comparative Study of Statistical, Machine Learning and Deep Learning Techniques. page 10.

# Appendix A

# Benchmark Implementation Details

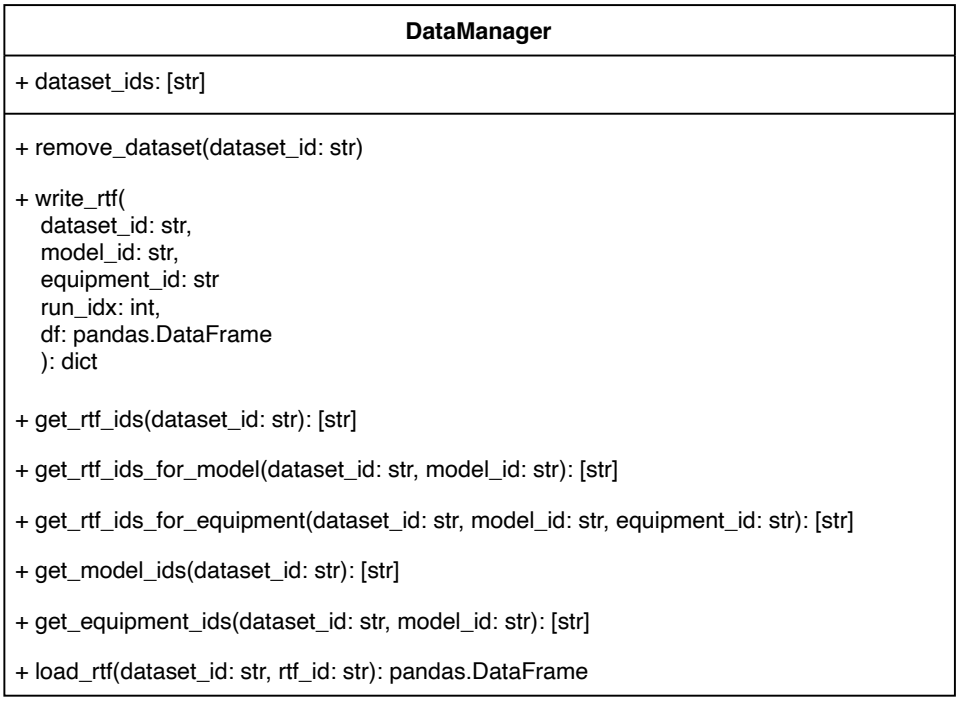



Figure A.1: UML of the DataManager Class

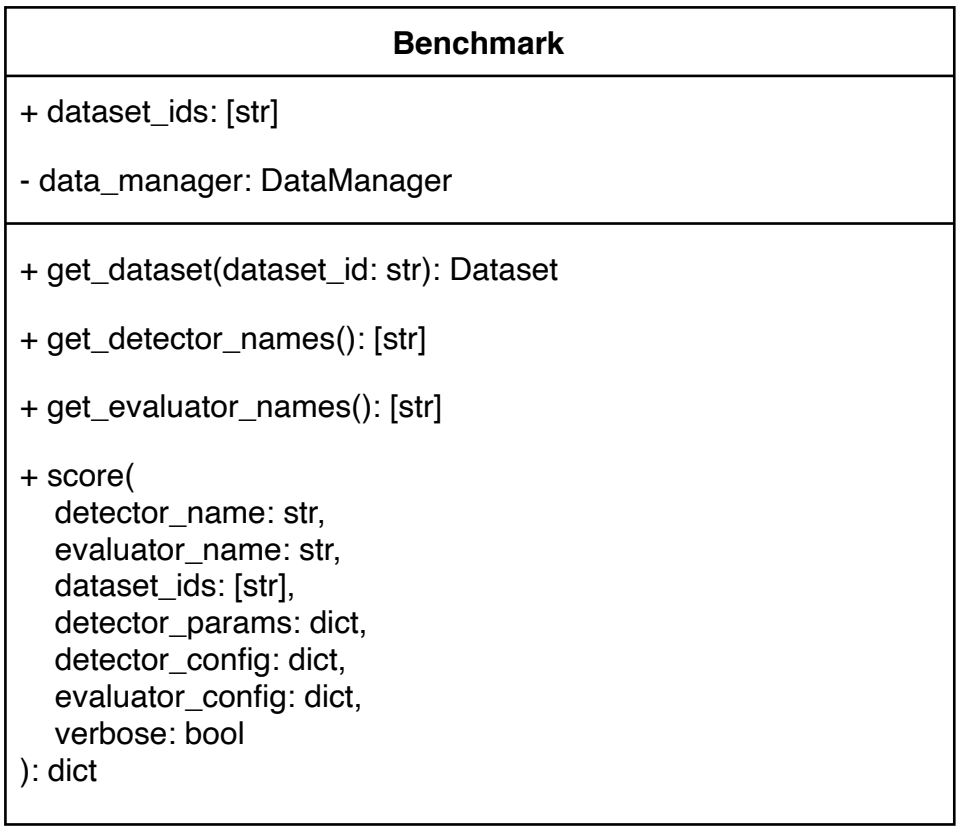



Figure A.2: UML of the Benchmark Class

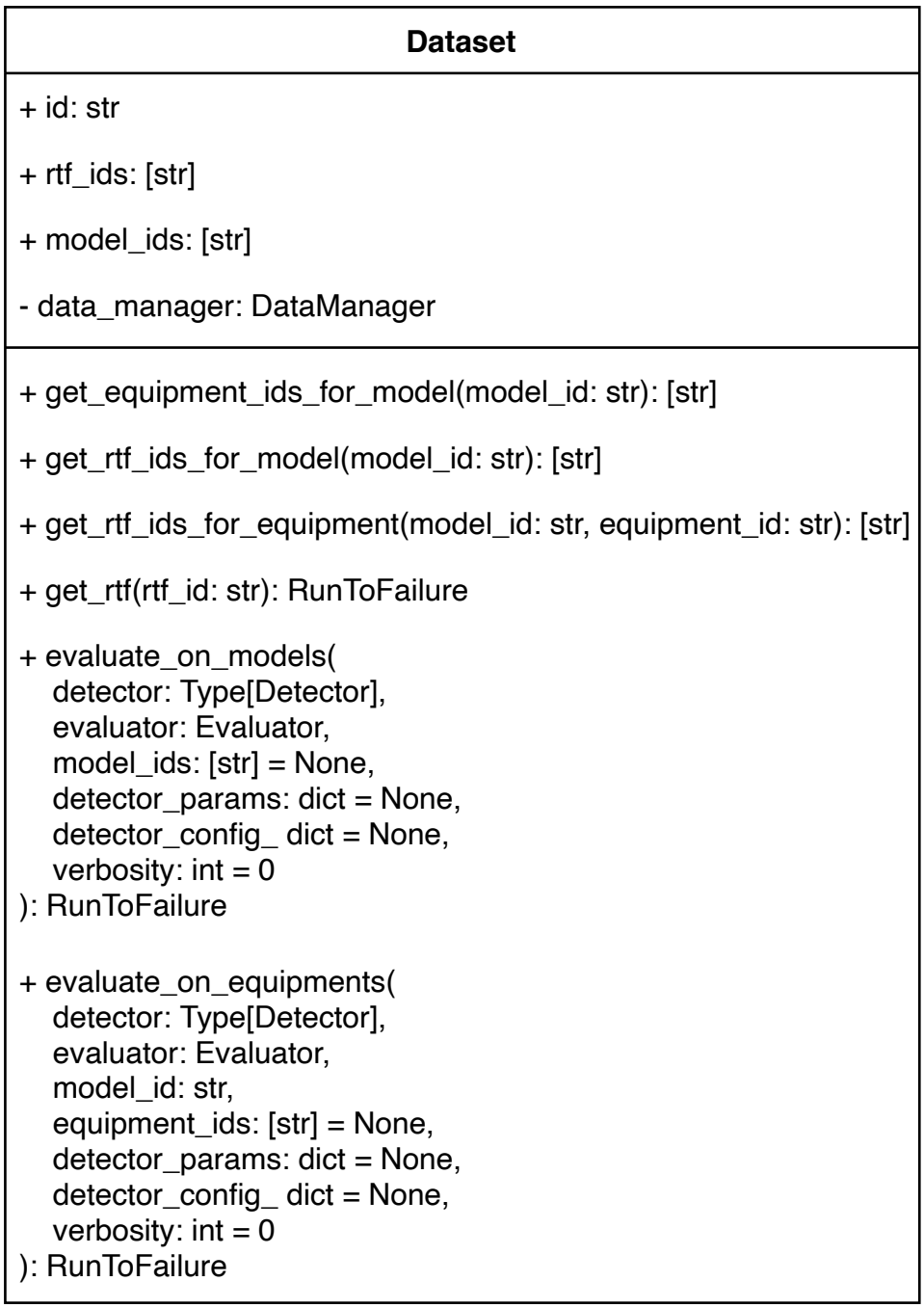



Figure A.3: UML of the Dataset Class

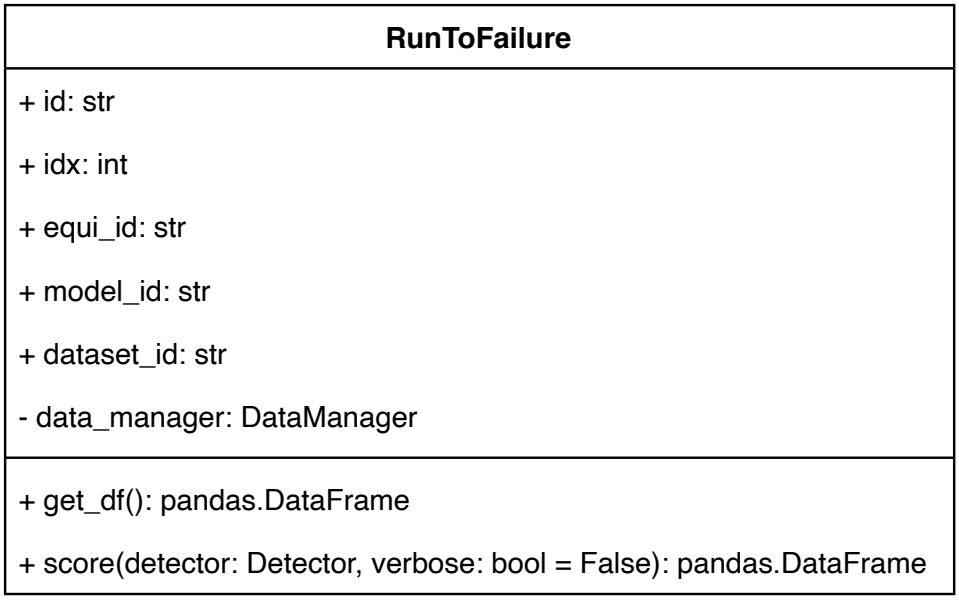



Figure A.4: UML of the RunToFailure Class

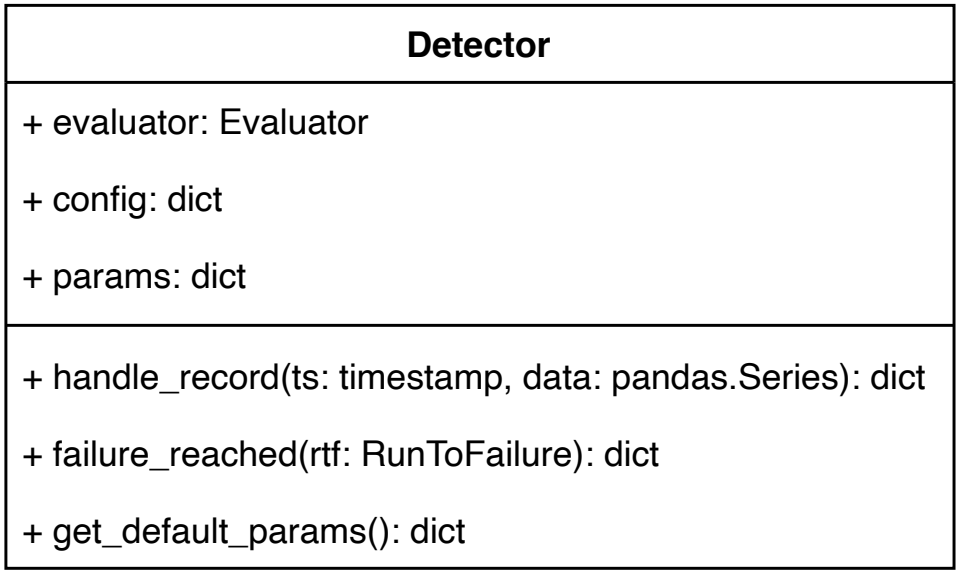



Figure A.5: UML of the Detector Class

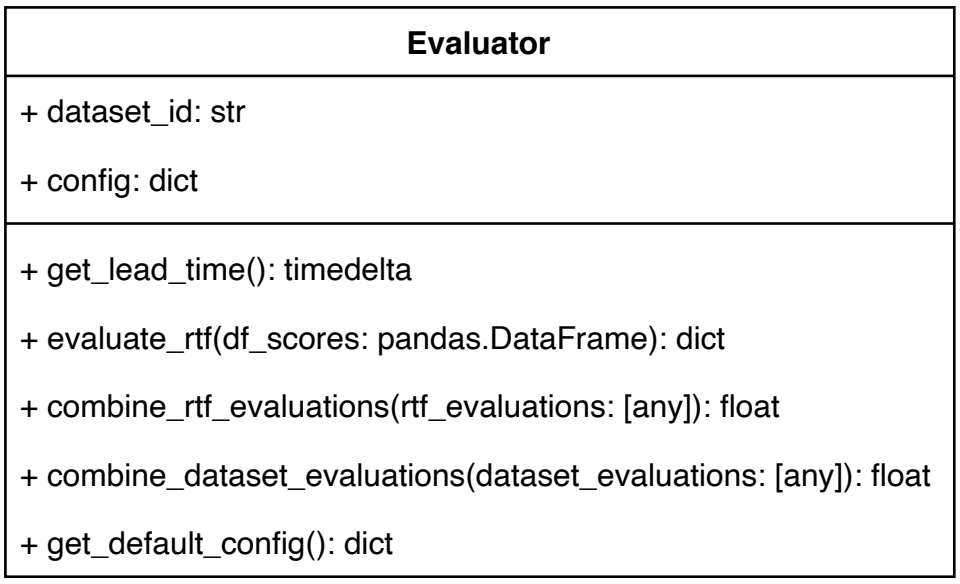



Figure A.6: UML of the Evaluator Class

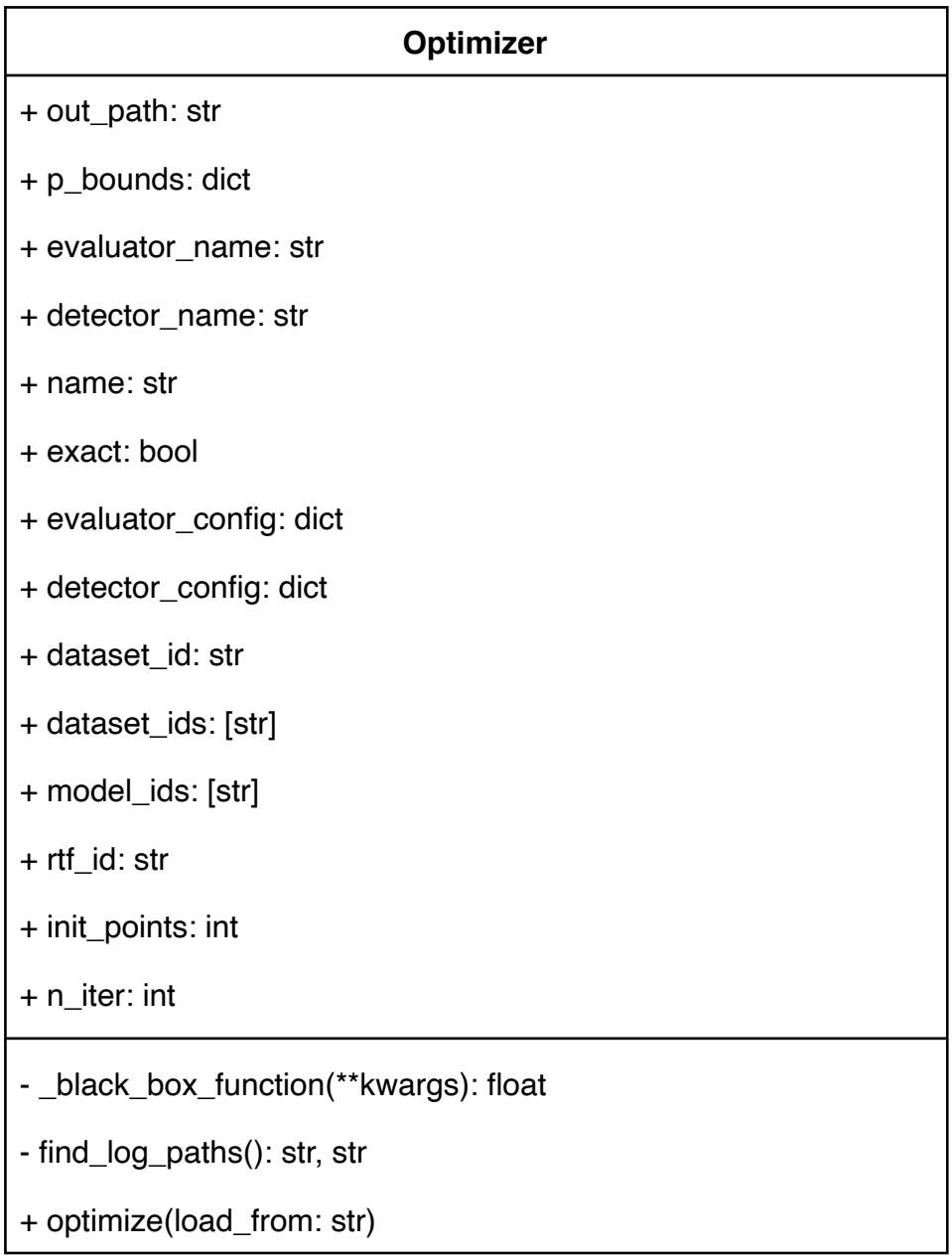



Figure A.7: UML of the Optimizer Class

# Appendix B

# Parameter Settings

| Algorithm | Parameter | Value |
|---|---|---|
| DateTime Encoder | timeOfDay | (21, 6.456) |
| RDSE | activeBits | 23 |
| | size | 400 |
| Spatial Pooler | boostStrength | 0.0 |
| | wrapAround | True |
| | columnDimensions | 1487 |
| | dutyCyclePeriod | 1017 |
| | minPctOverlapDutyCycle | 0.00090 |
| | localAreaDensity | 0 |
| | numActiveColumnsPerInhArea | 40 |
| | potentialPct | 0.92817 |
| | globalInhibition | True |
| | stimulusThreshold | 0 |
| | synPermActiveInc | 0.0038 |
| | synPermConnected | 0.2211 |
| | synPermInactiveDec | 0.00061 |
| Temporal Memory | activationThreshold | 14 |
| | cellsPerColumn | 32 |
| | connectedPermanence | 0.433 |
| | initialPermanence | 0.239 |
| | maxNewSynapseCount | 27 |
| | maxSegmentsPerCell | 161 |
| | maxSynapsesPerSegment | 141 |

| | | |
|---|---|---|
| | minThreshold | 13 |
| | permanenceDecrement | 0.00840 |
| | permanenceIncrement | 0.046 |
| | predictedSegmentDecrement | 0.00099 |
| Anomaly Likelihood | probationaryPct | 0.1079 |
| | reestimationPeriod | 100 |
| Threshold-based Detector | spatialTolerance | 0.041 |

Table B.1: Parameters for the univariate htm.core-based anomaly detection model

| Algorithm | Parameter | Value |
|---|---|---|
| HTM | shortProbationaryPeriod | 40 |
| | longProbationaryPeriod | 587 |
| | longSmoothingKernelSizeRatio | 0.7414 |
| | shortSmoothingKernelSizeRatio | 0.7156 |
| DSPOT | shortDepthRatio | 0.3311 |
| | longDepthRatio | 0.2813 |
| | shortProbationaryPeriodRatio | 0.6696 |
| | longProbationaryPeriodRatio | 0.4867 |
| | shortQ | 0.4973 |
| | longQ | 0.1570 |

Table B.2: Parameter settings for the multimodel architecture

| Algorithm | Parameter | Value |
|---|---|---|
| DSPOT | shortDepthRatio | 0.1425 |
| | longDepthRatio | 0.2567 |
| | shortProbationaryPeriodRatio | 0.1351 |
| | longProbationaryPeriodRatio | 0.7230 |
| | shortQ | 0.0849 |
| | longQ | 0.4382 |
| HTM | shortProbationaryPeriod | 91 |
| | longProbationaryPeriod | 4844 |
| DateTime Encoder | timeOfDay | (21, 6.456) |
| RDSE | activeBits | 23 |
| | size | 400 |
| Spatial Pooler | boostStrength | 0.0 |

| | | |
|---|---|---|
| | wrapAround | True |
| | columnDimensions | 7231 |
| | dutyCyclePeriod | 101 |
| | minPctOverlapDutyCycle | 0.3023 |
| | localAreaDensity | 0 |
| | numActiveColumnsPerInhArea | 77 |
| | potentialPct | 0.09233 |
| | globalInhibition | True |
| | stimulusThreshold | 93 |
| | synPermActiveInc | 0.3455 |
| | synPermConnected | 0.3967 |
| | synPermInactiveDec | 0.5388 |
| Temporal Memory | activationThreshold | 212 |
| | cellsPerColumn | 344 |
| | connectedPermanence | 0.2044 |
| | initialPermanence | 0.8781 |
| | maxNewSynapseCount | 32 |
| | maxSegmentsPerCell | 672 |
| | maxSynapsesPerSegment | 420 |
| | minThreshold | 281 |
| | permanenceDecrement | 0.1403 |
| | permanenceIncrement | 0.1981 |
| | predictedSegmentDecrement | 0.8007 |
| Anomaly Likelihood | probationaryPct | 0.1079 |
| | reestimationPeriod | 100 |
| Threshold-based Detector | spatialTolerance | 0.041 |

Table B.3: Parameter settings for the unimodel architecture

## Ehrenwörtliche Erklärung

Ich versichere, dass ich die beiliegende Masterarbeit ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen. Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Mannheim, den 31.08.2020 Unterschrift