University of Udine
Department of Mathematics, Computer Science and Physics
PhD Course in Computer Science, Mathematics and Physics

Ph.D. Thesis

Smart Sensing: Selection, Prediction and Monitoring

Candidate:
Andrea Urgolo

Supervisor:
Prof. Dr. Angelo Montanari

Co-Supervisors:
Dr. Andrea Brunello
Dr. Federico Pittino

Year 2023

Author's address:

Department of Mathematics, Computer Science and Physics
University of Udine
Via delle Scienze, 206
33100 Udine
Italy

There is no problem
that cannot be solved
by a 7-layer lasagna.

– *Gargiulo's mom (Inspector Coliandro)*

Abstract

A *sensor* is a device which is used to detect physical parameters of interest like temperature, pressure, or strain, performing the so called *sensing* process. This kind of device has been widely adopted in different fields such as aeronautics, automotive, security, logistics, health-care and more. The essential difference between a *smart sensor* and a standard sensor is its intelligence capability: smart sensors are able to capture and elaborate data from the environment while communicating and interacting with other systems in order to make predictions and find intelligent solutions based on the application needs. The first part of this thesis is focused on the problem of *sensor selection* in the context of *virtual sensing* of temperature in indoor environments, a topic of paramount importance which allows to increase the accuracy of the predictive models employed in the following phases by providing more informative data. In particular, virtual sensing refers to the process of estimating or predicting physical parameters without relying on physical sensors, using computational algorithms and predictive models to gather and analyze data for accurate predictions. We analyze the literature, propose and evaluate methodologies and solutions for sensor selection and placement based on *machine learning* techniques, including *evolutionary algorithms*. Thereafter, once determined which physical sensors to wield, the focus shifts to the actual methodology for virtual sensing strategies for the prediction of temperatures allowing to uniformly monitor uncovered or unreachable locations, reducing the sensors deployment costs and providing, at the same time, a fallback solution in case of sensor failures. For this purpose, we conduct a comprehensive assessment of different virtual sensing strategies including novel solutions proposed based on recurrent neural networks and graph neural networks able to effectively exploit spatio-temporal features. The methodologies considered so far are able to accurately complete the information coming from real physical sensors, allowing us to effectively carry out monitoring tasks such as anomaly or event detection. Therefore, the final part of this work looks at sensors from another, more formal, point of view. Specifically, it is devoted to the study and design of a framework aimed at pairing *monitoring* and *machine learning* techniques in order to detect, in a preemptive manner, critical behaviours of a system that could lead to a failure. This is done extracting interpretable properties, expressed in a given temporal logic formalism, from sensor data. The proposed framework is evaluated through an experimental assessment performed on benchmark datasets, and then compared to previous approaches from the literature.

Acknowledgments

Engaging in this PhD program has proven to be an immensely transformative and rewarding journey. I am sincerely grateful to all those who have contributed to the completion of this thesis. Their unwavering support, encouragement, and expertise have been vital in transforming ideas into reality. I will forever cherish the memories and knowledge gained from this collaborative endeavor.

First and foremost, I would like to express my heartfelt gratitude to my supervisor, Professor Angelo Montanari from the University of Udine, for his invaluable guidance and wisdom throughout these three remarkable years of my doctoral journey. His profound insights and unwavering support have been instrumental in shaping this thesis and my growth as a researcher. I am truly indebted to his scholarly tutelage and mentorship.

I would also like to extend my sincere appreciation to Andrea Brunello and Federico Pittino, my co-supervisors, for their invaluable contributions and constant encouragement. Their expertise and collaborative spirit have significantly enriched my research experience. I am immensely grateful for their insightful feedback and tireless dedication.

Furthermore, I would like to extend my gratitude to all the members of the Data Science and Automatic Verification Laboratory at the University of Udine. To Dario Della Monica, Nicola Saccomanno, Luca Geatti, and Gabriele Puppis, I am deeply thankful for their unwavering support and patience throughout the various stages of my research. Their expertise and willingness to assist have been invaluable in overcoming challenges and achieving the results presented in this thesis.

I am also deeply grateful to Silicon Austria Labs GmbH (SAL) for sponsoring my doctoral studies and providing me with a splendid opportunity to spend six memorable months in Graz and Villach. In particular, I would like to express my sincere appreciation to Ingo Pill and all the researchers at SAL for their invaluable assistance in acclimating to Austria. Their companionship, suggestions on places to visit, and, of course, the delightful taste of Kürbiskernöl have greatly enhanced my experience.

I would like to express my gratitude to Professor Paolo Vidoni and the research

group of statisticians at the University of Udine for their insightful consultations and invaluable advice. Their expertise and guidance in the statistical aspects of my research have been truly enlightening.

Finally, I cannot fail to mention the individuals with whom I shared an office space and countless cups of coffee. To Davide, Martina, Vittorio, Fernando, and Sebastiano, I am grateful for the laughter and stimulating discussions that made the research journey more enjoyable.

Contents

List of Figures

List of Tables

1

Introduction

Sensing is to grasp the nature, importance, or meaning of a physical phenomenon or stimulus. Apart from classic components, such as transducers, amplifiers, filters and compensators, a *smart sensor* differs from a traditional one as it has a data computing unit, a signal processing unit, and communication unit allowing it to be capable of collecting, processing, and communicating data with other devices [177]. Hence, it is important instrument for continuous monitoring and control action performing. Thanks to the improvements of nanotechnology and microelectronic technologies, increasingly complex and miniaturized smart sensors and devices are spreading on the market. These smart devices, which are an integral part of the Internet of Things (IoT) and Industry 4.0, are able to capture and elaborate data from the environment, communicate and interact with other systems and with the environment itself, making predictions and finding intelligent solutions based on the application needs [192]. Such devices may be used in different areas such as aeronautics, automotives, security, logistics, health-care, smart grid for intelligent energy production and consumption, predictive maintenance, industrial processes engineering and more [115, 189, 124, 83, 37, 192, 193]. More in general, what is proposed is typically aimed at achieving greater efficiency by optimizing management and processes, reducing the costs and the resources used [115].

Within the scope of this thesis, my primary objectives are to explore and propose comprehensive strategies, methodologies, and techniques for effective, flexible, and efficient environmental monitoring through smart sensor systems. Motivated by the advancements in nanotechnology and microelectronic technologies, as well as the increasing prevalence of Internet of Things (IoT) and Industry 4.0, my research aims to address challenges related to sensor placement, virtual sensing, system failure detection, and predictive maintenance. By optimizing the deployment of sensors, enhancing energy efficiency, and leveraging advanced machine learning solutions, I seek to contribute to the field by improving monitoring capabilities, reducing costs, and optimizing management and processes. To guide my investigation, I pose the

following key research questions: How can we determine the optimal placement of sensors to ensure uniform monitoring of physical phenomena? How can virtual sensing techniques be developed to predict physical quantities without relying on certain physical sensors? How can monitoring and machine learning techniques be integrated to detect critical system behaviors and enable predictive maintenance? By addressing these questions through an interdisciplinary approach, this thesis aims to contribute to the advancement of smart sensor systems and their applications in various domains.

Physical sensors are usually costly, in need of maintainance and sometimes unreliable [109]. Furthermore, optimizing energy consumption is a crucial aspect both to reduce the environmental impact given by the large scale usage of electronic devices, and because some systems have limited autonomy. Therefore, it is important to optimize energy consumption, limiting the number of engaged sensors while maintaining a good level of accuracy in the collected data and the intelligent task. The first obstacle to be tackled in order to deploy a sensors system is given by the choice of how to set an adequate number of sensors at the desire location to ensure a uniform monitoring of some physical phenomenon of interest. In [28] we deal with this problem in the case of temperature monitoring in an open space office. Here, a systematic analysis of several distance metrics that can be used to determine the best sensors on which to base temperature monitoring is performed. Then, following a genetic programming approach, we design a novel metric that non-linearly combines and synthesizes information brought by the considered distance metrics, outperforming their effectiveness. Thereafter, exploiting such a metric we propose an automatic and generic approach based on a weighted Borda count voting system to determine the best subset of sensors that are worth keeping. Borda count [19] is a widely adopted voting method that assigns points to candidates based on their ranking by voters, providing a fair and inclusive strategy to decision-making in voting processes. In the considered sensor selection scenario, it allows all the examined sensors to vote for candidate ones to select, and has been devised considering some important aspects. First, selected sensors may provide information that is relevant either locally or globally. Locally-relevant solutions will be a first choice only for just a niche of sensors, and they will rank poorly among the remaining voters. Globally-relevant solutions may not be the first choice, but they likely be an acceptable alternative option according to a vast majority. Anomalous sensors will be relevant only locally if not to themselves alone, while well-behaved sensors instead will usually be aligned with general trends and thus globally more relevant. Usually, sensors can surrogate each other depending on distance (proximity principle). Nevertheless, each sensor is usually predicted by itself better than by anything else, so it would likely vote

for itself if allowed to. This holds for all sensors, especially for anomalous ones, as they can rely only on themselves to predict their own troubled data. A voting system should prevent anomalous sensors to drive the election by design. Instead, it should foster the ability of different candidates to surrogate voters' data. Anomalous sensors will surrogate the others poorly, so they will likely lose the election.

With the aim of proposing and evaluating techniques capable of monitoring temperatures in uncovered or unreachable locations, we design and produced a set of simulation data by means of computational fluid dynamics algorithms for a generic room model, concerning different environmental and usage conditions, as well as different configurations in terms of arrangements of internal objects. In this scenario, starting from a sensors grid extended in 3 dimensions and sufficiently dense, we devised an optimal sensor placement strategy based on graph attention mechanisms, along with an algorithm capable of transferring the displacement to another simulated room with a different internal configuration.

Once the problem of sensor placement was addressed and the commissioning was done, we might focus on the issue of sensing physical quantities present in the environment, production processes or machinery. In many practical applications, obtaining the sensing data by placing the sensors at the optimal locations is difficult or even impossible, for this purpose, it may be necessary to implement *virtual sensing* solutions. Virtual sensing is a set of techniques to replace a subset of physical sensors by virtual ones, allowing the monitoring of unreachable locations, reducing the sensors deployment costs, and providing a fallback solution for sensor failures [107].

Hereafter, we take into account a real-world scenario of an indoor open space office dataset [26], and propose a black-box virtual sensing framework, capable of predicting temperatures exploiting spatio-temporal information, that, in principle, can be adapted to any indoor environment. More in detail, in [28] we proposed some solutions, including among all a long short-term memory recurrent neural networks-based model for the prediction of temperatures observed by physical sensors based on other sensors' data, and carried out a comparison with other approaches from the literature, including baseline methods, and some classical machine learning models evaluating also the reliability of the generated predictions.

The problem of monitoring temperatures in indoor environments, which is critical for several reasons, including satisfaction of comfort levels, energy efficiency, and safe temperature constraints, may require the devising of virtual sensing techniques capable of performing spatial interpolation, i.e., able to uniformly monitor each location of an examined environment. In this scenario, in order to evaluate the interpolation capacity of the assessed models, we focus on the aforementioned simulation dataset concerning different environmental and usage conditions for a generic

room model. The data observed from the sensor grid are exploited to evaluate the effectiveness of the analysed solutions. In detail, we propose deep learning approaches including a graph neural network model, capable of acting as a simulations approximant, representing information in order to learn physical characteristics of the measured phenomena. Here the spatial interpolation is possible thanks to the flexible structure of the graphs and exploiting the inductive learning capabilities of this model. This ensures it is possible to reuse models generated in the training phase, while adding and removing nodes (or edges) of the graph, or modifying their features. Subsequently, besides a systematic comparative concerning classical spatial interpolation approaches, we conducted a series of experiments aimed at verifying the ability of the proposed model to generalize on different simulated environmental conditions.

Once a sensors system has been made operational and efficient, regardless of whether it includes virtual or real sensors, it is able to provide a large amount of dynamic and heterogeneous data, as continuous streams collected over time and related to different kinds of statistical variables, such as categorical and numerical ones. This raw data can be analyzed in real time and processed to carry out event, anomaly or failure detection activities. Regardless of whether a commercial or private use is considered, this is a task that plays a fundamental role and allows the sensor system to be enhanced with control functions. The possible applications are many and vary, for example, from production process control, predictive maintenance of machinery and household appliances, to the detection of environmental comfort condition violations.

With this premise, the last part of this thesis is focused on system failure detection and predictive maintenance. In particular, on the basis of a set of values recorded by sensors, it is desired to recognize whether or not a production environment, a machine or a device shows an anomaly or an internal fault. The early detection of the latter can be useful, e.g., to prevent a failure of the entire system, a serious damage, a production block, or the occurrence of critical safety conditions. In the literature deep learning models have been exploited for these tasks with increasing success. Although these models offer good performances in terms of predictions, they hardly provide guarantees over their execution, a problem which is exacerbated by their lack of interpretability (i.e., they do not provide any qualitative understanding on the process leading from input variables to the final decision [154]). For this reasons, in many critical contexts, formal methods, which are able of ensuring the correct behavior of a system, are thus necessary. However, specifying in advance all the relevant properties and building a complete model of the system against which to check them is often out of reach in real-world scenarios. To over-

come these limitations, we proposed [24] a framework that resorts to monitoring (a lightweight verification technique that does not require an explicit model specification) and pairs it with machine learning, in order to automatically derive relevant properties, related to a bad behavior of the considered system, encoded by means of interpretable formulas expressed in a temporal logic formalism.

The extracted properties are stored in a monitoring pool. As we will see, the management of this pool raises a number of issues that were taken into account in the design phase. First of all, the validity of an originally well-predictive property may change over time, for example due to updates/upgrades performed on the monitored system. Therefore, it is necessary to handle the pool in a non-monotonic way, providing a mechanism for removing ineffective properties based on a validity score. In addition, in the monitoring pool there may be multiple properties showing a similar behavior against the incoming input data. In that case, it suffices to keep just a single representative of those formulas. Finally, the framework must be able to handle cases where inserted properties conflict.

The latter topic represents, at the same time, the culmination of the entire research path, and the starting point for further research directions. With this thesis, we propose a comprehensive set of strategies, methodologies and techniques for effective, flexibile, and efficient environmental monitoring through smart sensor systems. Furthermore, we present a general framework based on monitoring and learning techniques, applicable to different contexts and complex systems, which is able to exploit sensors and telemetry data in order to perform early failure detection through the extraction temporal logic properties.

Below we present a synopsis of the thesis structure. In Chapter 2, we provide background information on the relevant topics, including temporal data, machine learning, deep learning, evolutionary algorithms, and runtime verification techniques. Thereafter, in Chapter 3, we focus on the task of sensor selection introducing the application domains related to real-world and simulated indoor environments, proposing two solutions based on weighted Borda count and graph attention networks, and assessing their effectiveness. Considering the same settings, Chapter 4, , deals with the temperature prediction task, providing a comprehensive assessment of different virtual sensing strategies including baseline approaches, and novel solutions proposed based on recurrent neural networks and graph neural networks. In Chapter 5, we present a framework combining monitoring and machine learning techniques to perform failure detection and predictive maintenance tasks. The framework is then evaluated on different case studies widely adopted in the literature. Finally, we provide an overall assessment of the work done, together with future research directions.

2

Background

This chapter gives an overview of the main topics discussed in the thesis. Specifically, in Section 2.1 the notion of temporal data is introduced. Then, Section 2.2 focuses on machine learning, a subfield of artificial intelligence concerning different algorithms able to learn from data and make predictions without being explicitly programmed, and Section 2.3 on deep learning, which extends the capabilities of machine learning to extract complex features from data. This latter section introduces the notions of deep feedforward networks, recurrent neural networks, and graph neural networks. Section 2.4 discusses evolutionary algorithms, which are inspired by the process of natural selection and can be used for optimization problems. Here, genetic programming and NSGA-III multi-objective evolutionary algorithm are introduced. Finally, Section 2.5 explores runtime verification techniques, such as monitoring, a set of methods used to verify the correctness of a system while it is running. The application domains and case studies considered in this work will be presented progressively as they are addressed in the subsequent chapters.

2.1 Temporal data

Two common representations for temporal data are *time series* and *temporal sequences*. Time series are real-valued sequences of measurements taken at regular temporal intervals. A time series $X = \{x_1, x_2, \ldots, x_n\}$ for $T = \{t_1, t_2, \ldots, t_n\}$ is a discrete function having value x_1 for time t_1, x_2 for t_2, and so on. A time series might be *multivariate* or *univariate*. A multivariate time series is formed by more than one variable, whereas a univariate time series has only one underlying variable. Another distinction between time series is whether they are *stationary* or *nonstationary*. Stationary time series have constant mean and variance over time, whereas nonstationary time series have no discernible mean and can decrease or increase over time. Temporal sequences are discrete sequence of finite-domain values taken at regular or irregular time intervals. An example of temporal sequence could be a sequence of logged events, function calls, alarms, etc. It is worth noticing that also

a single event could be considered as a degenerate case of a temporal sequence with one time-stamped element.

In order to be fruitfully analyzed and processed, data must be appropriately treated to make them "clean" and suitable for the intended task of interest. This stage typically includes steps such as handling missing data [167], noise removal [122], normalization [59], feature engineering [191] and dimensionality reduction [38]. Given their strong dependence on the specific dataset and task, the latter will be described from time to time while proceeding with the following chapters of the thesis.

Within the scope of this study, as a normalization technique, we consider scaling the temporal data to a desired range. Specifically, we employ the method of *min-max scaling*, which transforms the data so that it falls within a predefined interval, typically between 0 and 1. Min-max scaling offers several benefits for the analysis of temporal data. Firstly, it standardizes the data, making it comparable across different features or variables. This standardization ensures that no single feature dominates the analysis based on its magnitude. Secondly, min-max scaling preserves the relative relationships among data points, allowing for meaningful comparisons and interpretations. Conversely, normalization for a normal distribution (e.g., *z-score normalization*), involves transforming the data to have a mean of 0 and a standard deviation of 1. In this case, the purpose of adapting data to a normal distribution is primarily relevant when statistical assumptions require the data to meet specific criteria, such as in tests that rely on distributional assumptions.

2.2 Machine learning

Machine learning is a branch of Artificial Intelligence focused on algorithms and statistical models able to learn from the experience automatically, hence, without any explicit programming. There exist two main categories of learning: *supervised* and *unsupervised* [16].

In supervised learning, the set of data employed in the training phase is made up of *instances* (or *examples*) labeled with ground truth target values (called *labels*). The objective of supervised learning is to learn how to predict the correct values for the target vector starting from a given input instances. In the case of *classification* problems, target values could consists in a finite number of discrete categories. Otherwise, in *regression* problems the target of the prediction is a numerical variable [4]. More formally, an instance is represented as a vector $\mathbf{x} \in \mathbb{R}^n$, where each of the n entries of the vector is called *feature* (or *predictor*). In classification tasks, the learning algorithm is usually asked to produce a function $f : \mathbb{R}^n \to \{1, \ldots, k\}$ mapping an input $\mathbf{x}$ to a category $y = f(\mathbf{x})$. Instead, in regression tasks models are

asked to learn a function with a numeric output $f : \mathbb{R}^n \to \mathbb{R}$.

On the other hand, unsupervised learning does not make usage of labels, but is rather focused on studying the data distribution, finding relations between variables, general patterns in the data, or groups of data instances [84].

In an intermediate position between supervised and unsupervised learning, lies *semi-supervised* learning. In this case, a model learns from both labeled and unlabeled data. By incorporating both types of data, this learning approach can improve the accuracy of a model and make it more robust, while reducing the reliance on labeled data. Due to the large amount of unlabeled data available in many domains, semi-supervised learning has been an active area of research in recent years. Some popular approaches in this field include *self-training*, *co-training*, and *graph-based methods* [39].

In machine learning, we often need to make predictions about new, unseen data based on the patterns observed in the data that we have. *Inductive learning* and *transductive learning* are two approaches to deal with this problem [175].

In more detail, inductive learning refers to the process of learning a general rule from a set of specific examples. The goal is to generalize from the observed examples to make accurate predictions on new, unseen ones.

Transductive learning, on the other hand, refers to the process of making predictions about specific, already known examples in the test set (i.e., the set of test instances), rather than generalizing to unseen ones. The goal of transductive learning is to make accurate predictions on the test set based on the observed patterns in the training set (i.e., the set of instances used in the training phase).

The capacity of a machine learning algorithm to perform effectively on new unseen cases that are not part of the training set is known as *generalization*. On the other hand, *overfitting* is a phenomena that happens when the algorithm is too closely fitted to a peculiar and limited set of instances, and fails to apply its knowledge to new data.

As we shall see in Chapter 4, both inductive and transductive learning have their own advantages and disadvantages, and the choice between the two depends on the specific problem and available data.

2.3 Deep learning

In traditional machine learning, an handcrafted mapping from raw input data to a feature space is often required to fed predictive models with proper input in order to pursue a learning task effectively. *Deep learning* differs from this classical form of "shallow" learning as is able to learn far higher degrees of hierarchical abstraction

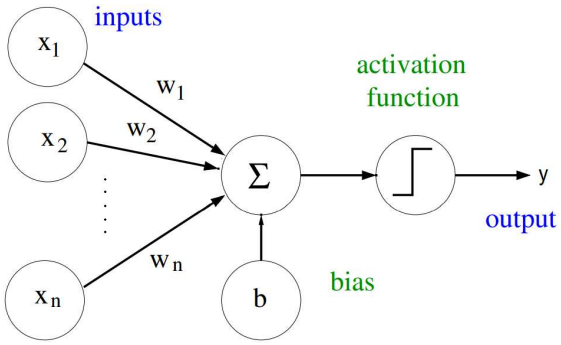


Figure 2.1: Model of an artificial neuron.

and representation [72]. The aim is to learn representations from the data with a progressive level of abstraction in another space (a latent space), where the problem become easier to solve.

In the following sections two of the main deep learning solutions used in this thesis are presented. In Section 2.3.1 deep feedforward networks are introduced along with some basic notions related to their employment. Finally, in Section 2.3.2 some recurrent neural networks architectures able to deal with sequential data are presented.

2.3.1 Deep feedforward networks

Deep feedforward networks, also called *feedforward neural networks* (FFNNs), or *multilayer perceptrons* (MLPs) [161] have served as the cornerstone of deep learning for decades. They are the fundamental building blocks of deep learning and could form complex architectures combining different non-linear transformations. Although they may not be regarded as pure deep learning models, the seminal role of MLPs in the development of deep learning cannot be overstated, as they have played a pivotal role in shaping the field and paving the way for subsequent advancements. This model, inspired by the biological brain, consists of simple processing units, the *artificial neurons*, organized in layers and connected by directed weighted edges.

As illustrated in Figure 2.1, artificial neurons compute a weighted sum of their inputs $w_1x_1 + w_2x_2 + \ \ldots \ + w_nx_n + b$. Then, the result is passed through an activation function ϕ. The purpose of activation functions is to, determine how the information flows from one neuron to the followings. This usually introduce non-linearities into the network, enabling it to approximate extremely non-linear functions [81]. The choice of this function depends on the architecture of the neural network and the specific task at hand. The most common activation functions

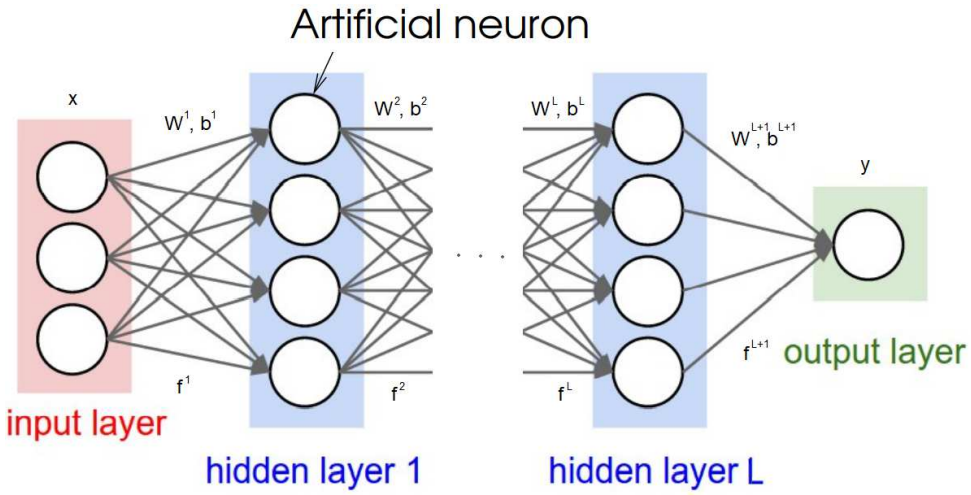


Figure 2.2: A basic feedforward neural network.

considered in the literature [137] are the sigmoid (*logistic*) function $\phi(x) = \frac{1}{1+exp(-x)}$, the hyperbolic tangent (*tanh*) function $\phi(x) = \frac{exp(x)-exp(-x)}{exp(x)+exp(-x)}$, the rectified linear unit (*ReLU*) function $\phi(x) = \max(0, x)$, and the leaky rectified linear unit (*Leaky ReLU*) function $\phi(x) = \max(\alpha x, x)$ where α is the negative input slope.

In Figure 2.2, an example of FFNN with an *input layer*, L intermediate layers, called *hidden layers*, and an *output layer*, is depicted. This network is able to calculate the output value $y = f(\mathbf{x}, \theta)$ starting from an input $\mathbf{x}$ and from parameters in $\theta = (\mathbf{W}^1, \mathbf{b}^1, \ldots, \mathbf{W}^{L+1}, \mathbf{b}^{L+1})$ applying the functions sequence $f_1, f_2, \ldots, f_{L+1}$ such that, for each k in $\{1, \ldots, k+1\}$, the output vector for layer k is $\mathbf{h}^k = f_k(\mathbf{h}^{k-1}, (\mathbf{W}^k, \mathbf{b}^k)) = \phi^k(\mathbf{W}^k\mathbf{h}^{k-1} + \mathbf{b}^k)$ with $\mathbf{h}^{k-1} \in \mathbb{R}^{n_{k-1}}$ output vector of level $k-1$, ϕ^k activtion function, $\mathbf{W}^k \in \mathbb{R}^{n_k \times n_{k-1}}$ weights matrix, $\mathbf{b}^k \in \mathbb{R}^{n_k}$ bias vector, n_k number of units (artificial neurons) of layer k, and $\mathbf{h}^0 = \mathbf{x}$ input vector.

After deciding on the network's architecture, the parameters θ must be estimated using a set of training samples. The estimation of θ is usually produced minimizing a *loss function* by employing a *gradient descent* algorithm [160]. The loss function quantifies how well the FFNN outputs are approximating the target values. Gradient descent is used to determine how to vary the weights and biases, in order to minimize the loss function. The variant of gradient descent primarily involved in modern neural networks training is the *stochastic gradient descent* algorithm. Here, as a first step, the parameters in θ are randomly initialized. This operation is crucial for FFNNs as it impacts convergence rate, stability, and the avoidance of poor local optima. An appropriate initialization breaks symmetry between units and controls variance to prevent vanishing or exploding gradients [69]. After the initialization of θ, for each training instance $\mathbf{x}^i$ and related target value y^i weights are updated

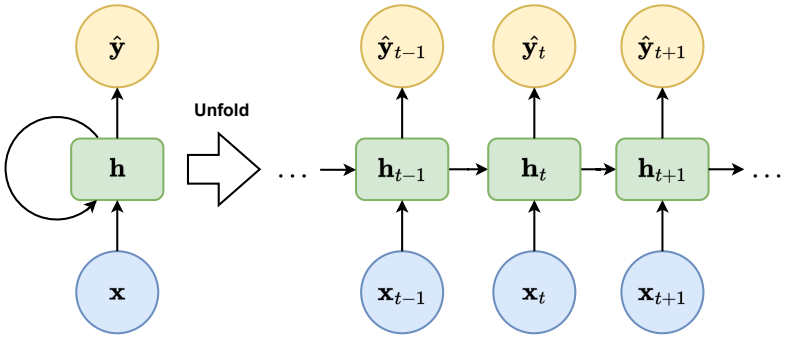


Figure 2.3: An unfolded vanilla recurrent neural network.

such that $\theta = \theta - \eta \nabla_\theta L(\theta, \mathbf{x}^i, y^i)$, where L is the loss function, $\nabla_\theta L(\theta, \mathbf{x}^i, y^i)$ its gradient obtained by deriving L on θ, and η is the *learning rate*. Instead of computing every gradient value associated to each parameter, the propagation of the gradients from the output layer, to the preceding ones is carried out through the *back-propagation* algorithm (introduced in [162]) taking advantage of the fact that all operations used in the network are differentiable. Furthermore, it is important to point out that the calibration of the learning rate η is of paramount importance for the convergence of the algorithm: if too small, the optimization can be blocked on a local minimum; conversely, if too large, the network will oscillate around an optimum. In order to reduce this oscillation effect and accelerate the convergence process, variations of the optimization algorithm, like *Nesterov accelerated gradient* (NAG) [134], *root mean square propagation* (RMSProp) [78] or *adaptive moment estimation* (Adam)[94], have been proposed in the literature.

Finally, It is worth noticing that, in order to avoid the effects of noisy or uninformative inputs, the gradient is not computed for each singular example, but on an entire subset of the training set, called *batch*, taken at random without replacement.

2.3.2 Recurrent neural networks

Feedforward neural networks have some limitations when they have to process temporal data. In fact, considering a sequence of data supplied as input, each of its elements is processed by a network independently of the position occupied within the sequence itself. Hence, temporal dependencies are implicitly ignored by this kind of models. To deal with this kind of data, feedforward neural networks have been extended in order to include feedback connections. Those kind of networks are called *recurrent neural networks* (RNNs) [168].

RNNs can learn from sequences and produce a sequence of states and outputs. In Figure 2.3 an example of "vanilla" RNN architecture is shown. As can be seen

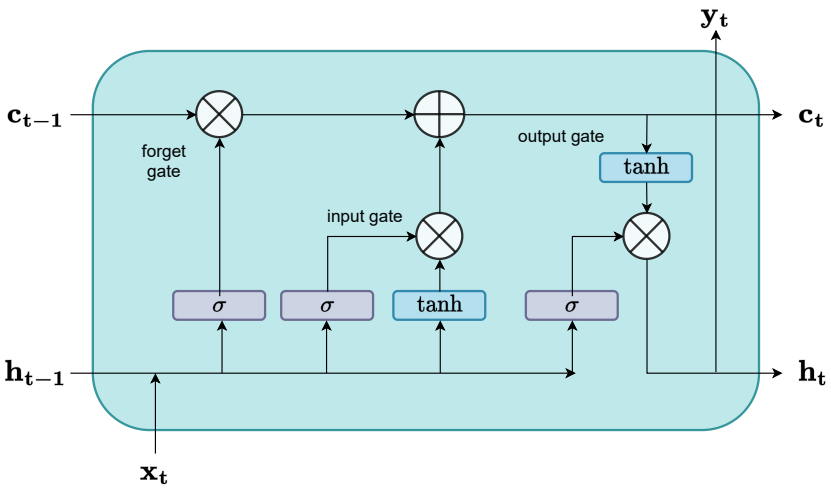


Figure 2.4: A long short-term memory recurrent neural network cell.

from the unfolded representation of the network on the right, this RNN processes a sequence by iterating through each of its elements, keeping a state containing information on what it has seen so far.

In the training phase, the gradient is propagated backward through all the sequence of states generated by the input. This process, called *back-propagation through time* [182], is the RNN counterpart for FFNN's back-propagation algorithm. In this settings, architecture like vanilla RNN could experience problems due to the *gradient vanishing* effect. The latter occurs in the presence of long sequences of input data and small weights in the network, making the gradient shrink at each step until it disappears. In fact, as could happen with very deep FFNNs made up of several layers, long products of matrices can lead to vanishing (or exploding) gradients. This cause RNN to be biased towards capturing only short-term relations and, consequently, to loose information on data seen several timesteps before. As we will see in the following part, A possible solutions to this issue, is the adoption of gated architectures like *gated recurrent unit* (GRU) [44] or *long short-term memory* (LSTM) [79].

Long short-term memory RNNs

The Long Short-Term Memory architecture, which is a special kind of RNN architecture, was created in order to enable the learning of long time dependencies from input sequences. This is done, thanks to a special cell architecture as depicted in Figure 2.4. As we can see, here two types of state are provided: $\mathbf{h}_t$, which is the short-

term state exposed to the outside, and $\mathbf{c}_t$, the internal long-term state of the cell. The core components of this architecture are the gates which control the flow of information learning what to store and what to throw away from the states and from the input data. In detail, given the parameters $\theta = (\mathbf{W}_{xf}, \mathbf{b}_f, \mathbf{W}_{xi}, \mathbf{b}_i, \mathbf{W}_{xo}, \mathbf{b}_o, \mathbf{W}_{xg}, \mathbf{b}_g,)$ an LSTM cell has three gates:

- the forget gate $\mathbf{f}_t$, which learns what part of the previous state $\mathbf{c}_{t-1}$ should be erased, and is defined by the function $\mathbf{f}_t = \sigma(\mathbf{W}_{hf}\mathbf{h}_{t-1} + \mathbf{W}_{xf}\mathbf{x}_t + \mathbf{b}_f)$;
- the input gate $\mathbf{i}_t$, which determines what relevant new information store in the long-term state $\mathbf{c}_t$, and is defined by the function $\mathbf{i}_t = \sigma(\mathbf{W}_{hi}\mathbf{h}_{t-1}+\mathbf{W}_{xi}\mathbf{x}_t+\mathbf{b}_i)$;
- the output gate $\mathbf{o}_t$, controlling what information encoded in the new long-term state $\mathbf{c}_t$ should be read and processed as output and as new value for $\mathbf{h}_t$; this gate is defined by the function $\mathbf{o}_t = \sigma(\mathbf{W}_{ho}\mathbf{h}_{t-1} + \mathbf{W}_{xo}\mathbf{x}_t + \mathbf{b}_o)$.

The value of the new long-term state is computed as $\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \mathbf{g}_t$, where $\mathbf{g}_t = tanh(\mathbf{W}_{hg}\mathbf{h}_{t-1} + \mathbf{W}_{xg}\mathbf{x}_t + \mathbf{b}_g)$ is the new candidate state and $\odot$ is the element-wise multiplication operator. Finally, the short-term state is updated as $\mathbf{h}_t = \mathbf{o}_t \odot tanh(\mathbf{c}_t)$.

It is worth noticing that the forget gate allows to completely filter or keep intact the information coming from the previous state $\mathbf{c}_{t-1}$. Thus, guaranteeing a theoretically back-propagation through time with uninterrupted gradient computation flow, LSTM RNNs can potentially learn long time dependencies related to input sequences. Considering the large amount of parameters in θ, the training phase can be computationally intensive. Hence, lighter architectures, with only a single state and fewer gates, such as RNNs with GRU cells, have been proposed in the literature [44].

2.3.3 Graph neural networks

In recent years, *graph neural networks* (GNNs) have emerged as a popular class of deep learning models for learning representations of graph-structured data. A graph is a mathematical structure that consists of a set of vertices or nodes connected by edges. Graphs can be used to represent a wide range of complex systems, such as social networks, biological molecules, and natural language sentences. GNNs extend the idea of *convolutional neural networks* (CNNs) [104] to graphs, enabling us to learn node and graph-level representations that capture the structural information of the input graphs.

Formally, a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is defined as a set of nodes (vertices) $\mathcal{V}$ and a set of edges $\mathcal{E}$, where each edge connects two nodes. The edges can be either directed

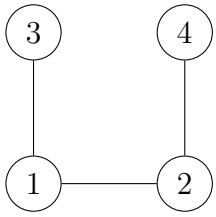


Figure 2.5: An example of undirected graph

or undirected, and can have weights that represent the strength of the relationship between the nodes.

The adjacency matrix $\mathbf{A}$ is a binary or weighted matrix that represents the connections between the nodes in the graph. For an undirected graph, the adjacency matrix is symmetric, and the diagonal entries represent the self-loops of each node. Formally, the adjacency matrix is defined as:

$$\mathbf{A}_{i,j} = \begin{cases} 1 & \text{if } (i,j) \in \mathcal{E} \\ 0 & \text{otherwise} \end{cases} \tag{2.1}$$

It is worth noticing that, for a weighted graph, the adjacency matrix may contains arbitrary real-values rather than $\{0, 1\}$.

As an example, considering the undirected graph in Figure 2.5 with nodes $\mathcal{V} = \{1, 2, 3, 4\}$ and edges $\mathcal{E} = \{(1, 2), (1, 3), (2, 4)\}$, the corresponding adjacency matrix for this graph is

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0. \end{bmatrix} \tag{2.2}$$

In graph theory, each vertex and edge can be associated with a representation that captures some of its characteristics. In particular, we can define *node representations* and *edge representations*. A node representation $\mathbf{h}_i$ is a vector that captures the properties of the vertex i in the graph. The exact definition of $\mathbf{h}_i$ depends on the application and can vary widely. For example, in machine learning tasks on graphs, node representations may encode information about the node's attributes, its neighbors, or its position in the graph. An edge representation $\mathbf{h}_{i,j}$ is a vector that captures the properties of the edge between vertices i and j. Edge representations can also vary depending on the application. For example, in some cases, $h_{i,j}$ may represent the weight or distance of the edge, or a more complex feature vector that captures the relationship between the two vertices. The adjacency matrix $\mathbf{A}$ can be

seen as an example of an edge representation that encodes the binary existence of edges in the graph.

GNNs can be broadly categorized into two types: *spectral-based* and *spatial-based* [75]. Spectral-based GNNs are founded on the spectral graph theory and rely on the eigenvalues and eigenvectors of the graph Laplacian matrix to define filters that operate on the graph. Spatial-based GNNs, on the other hand, are based on the message-passing paradigm, where each node aggregates information from its neighboring nodes and updates its representation accordingly. In our study, we will focus on spatial-based approaches and, in particular, on message passing neural networks.

Message passing neural networks

Message passing neural networks (MPNNs) [68] are a general framework for GNNs that use message passing to update node representations. In an MPNN, the message passing scheme is explicitly defined as a set of update rules that determine how each node's representation is updated based on messages received from its neighbors. The update rules can be formulated as follows:

$$\mathbf{m}_i^{(t)} = \sum_{j \in \mathcal{N}(i)} M^{(t)}(\mathbf{h}_i^{(t-1)}, \mathbf{h}_j^{(t-1)}, \mathbf{e}_{i,j})$$
$$\mathbf{h}_i^{(t)} = U^{(t)}(\mathbf{h}_i^{(t-1)}, \mathbf{m}_i^{(t)})$$

where $\mathbf{h}_i^{(t-1)}$ represents the node representation of node i at iteration $t-1$ and $\mathbf{e}_{i,j}$ the edge representation between nodes i and j, $\mathbf{m}_i^{(t)}$ is the message received by node i at iteration t, $\mathcal{N}(i)$ is the set of neighbours of node i, $M^{(t)}$ is a message function that determines how the message is computed based on the current node representations and the edge features, and $U^{(t)}$ is an update function that computes the new node representation based on the previous representation and the received messages.

There exist several specific models of MPNNs, which are defined by appropriately choosing which message function and update function should be employed. Specifically, in Chapters 3 and 4, we will adopt a particular type of MPNNs, namely graph attention neural networks (GATs) [176].

2.4 Evolutionary algorithm

Evolutionary algorithms (EAs) are population-based metaheuristics inspired by the process of biological evolution and genetics, that excel in the solution of combinato-

rial optimization problems [57]. Differently from classic random search, EAs make use of historical information to direct the search into the most promising regions of the search space.

In nature, a population of individuals tends to evolve to adapt to its environment. Similarly, EAs are characterized by a population, where each individual represents a candidate solution to a given optimization problem; each solution is evaluated with respect to its degree of "adaptation" to the problem through a single- or multi-objective *fitness* function.

The EA population iteratively goes through a series of *generations*. At each generation, individuals chosen by a *selection strategy* undergo a process of reproduction. Such a selection strategy is the fundamental factor that distinguishes one evolutionary-based approach from another, although, typically, individuals with high degree of adaptation are more likely to be chosen (*elitism*). In this way, the elements of the population iteratively evolve toward better solutions. Reproduction involves the application, with a certain degree of probability, of suitable *crossover* and *mutation* operators. As a result, an offspring is generated, which is finally merged with the previous population, and the cycle repeats until a stopping condition is met, e.g., based on a given fitness threshold.

Crossover is the EA counterpart of natural reproduction, by which the characteristics of two individuals are combined by generating one or two offspring. As a general rule, a high crossover probability tends to pull the population towards a local minimum or maximum. Mutation applies random changes to the encoding of the selected solution, with the goal of maintaining genetic diversity in the individuals; it prevents premature convergence of the algorithm to a local optimum, thus allowing it to explore the search space more broadly.

Evolutionary approaches have a tendency to develop solutions that are as good as they can be for the collection of examples against which the fitness function is measured, without taking into account the performances on potential new cases. As a consequence, generalization in evolutionary computation has been acknowledged as a significant open issue, and numerous efforts are being undertaken to address this problem [71, 48].

2.4.1 Genetic programming

In this work, we deal with a specific kind of optimization task, that is, *genetic programming* (GP). Such a technique evolves programs starting from a population of random solutions [146]. Each individual is encoded by means of a computation tree, where each leaf represents an input value (either a variable or a constant) and internal nodes encode operators applied over such values. In GP input val-

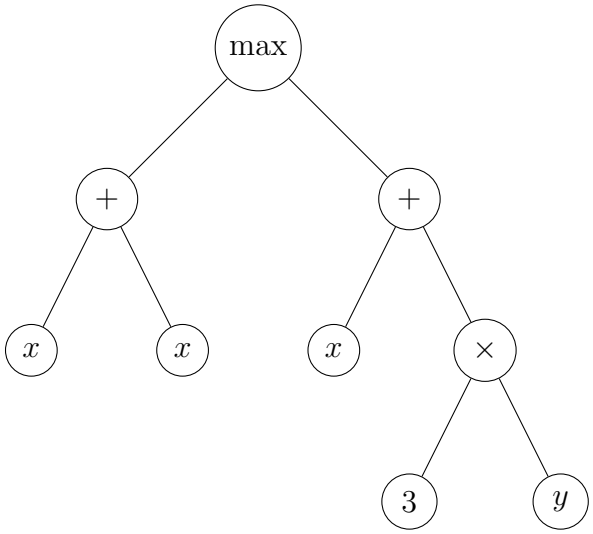


Figure 2.6: Genetic programming syntax tree representing the expression $\max(x + x, x + 3 * y)$.

ues are called *terminals*, while internal nodes *functions*. Joining the set of allowed terminals with that of functions we obtain the *primitive set* of the GP system. Furthermore, GP programs can be classified in two main kinds [128]: *loosely typed GP*, in which primitives' arguments can be any element of the primitive set; *strongly typed GP*, in which a specific type is assigned to every primitive, and the input type of each primitive argument must match the type of its child primitive. The output value is generated by the primitive encoded in the root. Typical crossover/mutation operations applied on computation trees are subtree exchange and node/leaf addition/removal/replacement. As an example, the program in Figure 2.6 represents the function $\max(x + x, x + 3 * y)$. Here the set of functions is $\{+, *, \max\}$, while the set of terminals is $\{x, y, 3\}$.

Finally, it is important to notice that through GP it is also possible to represent properties encoded by means of formulas expressed in some logical formalism. In this case, the computation trees take the place of syntax trees of formulas, and their evaluation will be obtained according to a semantics defined for the adopted logic.

2.4.2 NSGA-III multi-objective evolutionary algorithm

As we shall see, a generalization technique may also be implemented by employing two or more objective functions. EAs designed to solve a group of minimization/maximization problems for a tuple of n functions $f = \langle f_1, \ldots, f_n \rangle$ are called *multi-objective* EAs. A set $\mathcal{F}$ of solutions for an n-objective problem with fitness function $f = \langle f_1, \ldots, f_n \rangle$ is said to be *non-dominated* if and only if for each $x \in \mathcal{F}$,

there exists no $y \in \mathcal{F}$ such that (i) $f_i(y)$ improves $f_i(x)$ for some i, with $1 \leq i \leq n$, and (ii) for all j, with $1 \leq j \leq n$ and $j \neq i$, $f_j(x)$ does not improve $f_j(y)$. The set of all non-dominated solutions is called *Pareto front*.

NSGA-III (Non-dominated Sorting Genetic Algorithm) [50] is a multi-objetive optimization methodology which aims to solve multi-objective optimization problems efficiently and with the ability of distributing population members over the entire Pareto front (i.e., preserving its *diversity*), while giving to best solutions better chances of being carried over to subsequent generations (i.e., ensuring *elitism*).

In detail, NSGA-III is an extension of NSGA-II [51], whose peculiarity lies in the selection strategy. This latter is applied after the offspring generation phase in which the population has doubled in size. Here the combined (current and offspring) population of size $2*N$ is grouped into fronts in $\{\mathcal{F}_1, \mathcal{F}_2, \dots\}$ such that $\mathcal{F}_i$ is a Pareto front for the set $\cup_{j>i}\mathcal{F}_j$. Then, the selected population $\mathcal{F}'$ is such that $\mathcal{F}' = \cup_{i<l}\mathcal{F}_i$, with l defined as $\arg\min_l(|\cup_{i\leq l}\mathcal{F}_i| > N)$ (i.e., solutions with a higher non-dominant rank are preferred following an elitist criterion). The remaining $N - |\mathcal{F}'|$ solutions are chosen from $\mathcal{F}_l$ applying a selection process based on reference points. These points are widely and uniformly distributed on the normalized hyperplane inherent to the optimization objectives of the problem addressed by the algorithm. With the aim of preserving the diversity and distribution of the new population, this process emphasizes the selection of remaining solutions from $\mathcal{F}_l$ which are associated with every reference point by means of a distance function giving priority to individuals covering a greater number of reference points (niching strategy).

2.5 Runtime verification techniques

Runtime verification is the branch of computer science that deals with the research, development, and application of verification techniques capable of checking that a *run* of a system under scrutiny meets or violates certain correctness properties. This kind of techniques arose as a lightweight complement of more traditional verification techniques such as model checking [46] and testing [129]. A system run is defined as a possibly infinite sequence of system states. Formally, a run is considered to be a *word* (or a *trace*), and a system execution is a finite prefix of a run, hence a finite word (or trace).

In model checking, an exhaustive analysis is performed typically on finite-state systems, and the satisfaction or violation of a property is established by exploring every possible behavior. On the contrary, runtime verification allows to detect satisfaction of violation of a property by analyzing a single behavior (a run) of the system. Furthermore, in model checking a suitable model of the system to be checked must

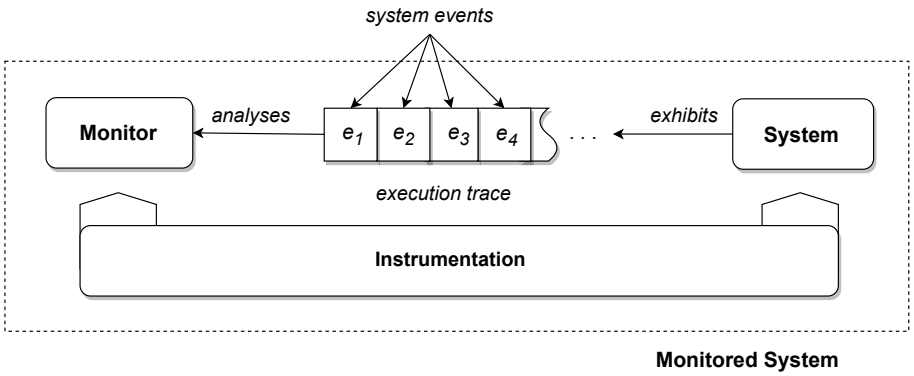


Figure 2.7: Example of monitoring setup.

be constructed. However, the inherent complexity of system's components and of their interactions could raise a state explosion problem, and it may be difficult to derive a complete model of the system against which to check the properties of interest. On the other hand, runtime verification is applicable to black-box systems for which no system model is at hand and, considering a single run, does usually not yield any memory problems. Nevertheless, the real system under analysis might behave in a slightly different way than how model predicts (e.g., because it strongly depends on the environment). In this scenario, runtime verification can be paired with model checking to double check behaviors or properties that have been already statically proved or tested.

Testing techniques, instead, require a test suite consisting of a finite set of finite input-output sequences. Then, the execution of these test cases checks whether the output of the system matches what was expected when the input sequence is fed to the system under test. In a sense, runtime verification can be seen as a continuous testing process. This latter, however, does not require a test suite as the inputs are generated directly from the execution of the system itself.

2.5.1 Monitoring

Monitoring [106] is a runtime verification techniques that is gaining much interest in the realm of formal methods for automated system verification. The basic principle on which monitoring is built are the ability of reaching a verdict by just observing a finite prefix of a single execution trace (a finite trace) and the fact that, once a verdict is reached, it is irrevocable and it is a guarantee that the system satisfies or violates the property, independently of all the other possible (unobserved) behaviors it might exhibit afterwards. This latter ensures the soundness of a monitor and makes such a technique naturally applicable to data streaming contexts.

A typical monitoring setup is depicted in Figure 2.7, and it consists of three main components: the system under scrutiny, the monitor and the *instrumentation* mechanism. A *monitor* is defined [135] as a computational entity executed along side a system so as to observe its runtime execution and possibly determine whether a property φ is satisfied or violated from the exhibited behavior. When a sufficiently long system execution trace is observed, a monitor may reach a verdict (e.g., acceptance or rejection). This verdict is assumed to be definite, as it cannot be retracted or revised. *Instrumentation* mechanism typically determines what aspects of the system behavior are notified to the monitor and also dictates how interaction between the components is carried out.

Monitors can be divided into two main classes, namely *online* and *offline* monitors. Online monitors are executed while the system is in operation and must be able to receive notifications about relevant events occurring in the system under scrutiny and make a decision on the basis of the information collected so far in an incremental fashion. Instead, offline monitors analyse relevant system events simulated or recorded as an execution trace inside a permanent data store after the system has finished running. Since in offline monitoring the execution of the monitor is independent of that of the system, an offline analysis is less intrusive, thus causing less runtime overhead. By contrast, online monitoring is capable of making early detections often exploited by runtime adaptation tools [34, 35] to solve or mitigate potential failures.

The simultaneous execution of the system and the monitor may be performed in different ways. In the case of *synchronous* online monitoring, every time the system generates an event, it waits for the monitor to process it before proceeding with its execution. Conversely, *asynchronous* online monitoring detaches the execution of the monitor from that of the system. This approach is less intrusive and typically leads to lower overheads, but may still yield a degree of late detections. Due to this, hybrid approaches [31] that fall on the spectrum in between these two are used to obtain the best of both worlds.

As we shall see in Section 5.2, although monitoring is a lightweight technique, the gain in efficiency is paid in terms of expressivity. Therefore, monitorable properties are a subset of the class of specifications that can be expressed in common temporal formalisms used for automated verification. Although monitoring has been mainly investigated in the context of linear time temporal logics[77], notable efforts have been devoted to studying monitoring of branching-time properties. A comparison between the two settings can be found in [1], and it is beyond the scope of this thesis, which focuses on the monitoring of linear time temporal properties.

3

Selection

In recent years, the development of sensor systems and their widespread availability have increased the potential for monitoring indoor environments. However, as discussed in Chapter 1, deploying and maintaining a large number of sensors can be expensive, and the sheer volume of data produced by such a system can pose significant challenges for data processing and analysis. Sensor selection has emerged as a promising practice to these challenges. By selecting a subset of sensors from a larger system, the cost of deploying and maintaining the system can be reduced while still monitoring the environment in an accurate and uniform manner. In addition, as we shall see in Chapter 4, virtual sensing techniques can be leveraged to complement the information grasped by physical sensors and provide a more complete picture of the environment.

This chapter focuses on the sensor selection task considering two distinct scenarios: the selection of sensors in real-world indoor environments and the placement of sensors in simulated indoor environments. The main goal is to propose solutions that, considering the task of temperature monitoring, maximize the efficiency and effectiveness of a sensor system while minimizing costs. Here, the primary objective of our research is to identify the relevant sensors for data collection rather than selecting specific features or attributes of the data itself. By focusing on sensor selection techniques that directly address the goal of selecting sensors, we can gain a clearer and more direct understanding of the performance of the proposed approaches. While a comparison with feature selection techniques may be interesting, it might be more meaningful and relevant to compare sensor selection techniques with other specific alternatives tailored to the problem domain taking also into account aspects related to sensor ranking and explainability.

After an introduction to the problem domain, in Section 3.1 we propose a novel procedure for sensor selection based on the weighted Borda count method. We demonstrate the effectiveness of this approach through a series of experiments conducted in a real-world indoor environment comparing with a brute force solution. Afterwards, in Section 3.2, we propose a solution based on graph attention neural

networks for sensor placement through sensor selection in simulated indoor environments. We conclude the latter section comparing the performance of our proposed solutions to each other and to a baseline of random sensor combinations. Our experimental results indicate that the method based on graph attention networks outperformed the weighted Borda count one in the considered scenario. Furthermore, both proposed solutions outperform the baseline, proving their ability to provide accurate monitoring of the environment while reducing costs.

3.1 Sensor selection in real-world indoor eviron-ments

The problem of sensor selection refers to the task of selecting a subset of sensors from a larger pool of available ones to optimally acquire information from an environment or a target system. In order to address this problem, different approaches have been explored in the literature in various domains. In [130], an optimal sensor selection and fusion solution is proposed for the case of an heat exchanger fouling diagnosis in aerospace systems. It is based on the minimum Redundancy Maximum Relevance (mRMR) algorithm, and it can be applied only for classification tasks with discretized features. Another sensor selection approach for optimal Fault Detection and Isolation (FDI) tests in complex systems is presented in [141]. In this case, steady-state or dynamic models are assumed to be available, and the estimation is based on their contribution to information gain through Hellinger distance (HD) and Kullback–Leibler divergence (KLD).

In the context of mobile crowd sensing (MCS) systems, which leverage a public crowd equipped with various mobile devices for large scale sensing tasks, a solution based on reverse combinatorial auctions is proposed in [87], that integrates the concepts of social welfare, quality of information, and cost required by each single user to provide an observation, in order to select an optimal subset of users from whom it is convenient to request data in exchange of a reward. This approach is not applicable to our case, as both the quality level of the sensors and the cost of each observation are the same.

A different class of problems where sensor selection techniques can be exploited is that of sensor scheduling problems, where one or more sensors have to be selected at every time step. As an example, in the domain of linear dynamical systems, a greedy algorithm for sensor scheduling based on submodular error functions is described in [86]. As another example, in the context of active robotic mapping, a technique to prune the search tree of all possible sensor schedules, based on a weighted function of the error covariances related to the state estimates, is illustrated in [178]. Lastly, an

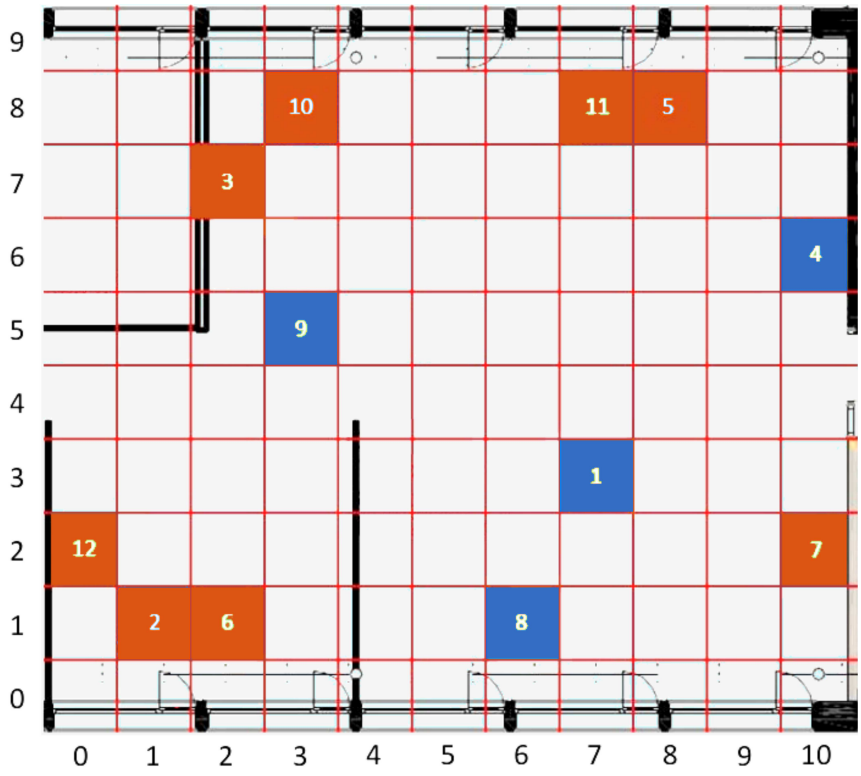


Figure 3.1: Location of the sensors in the considered premise. The blue cells represent the reference sensors that will be selected during the sensors selection phase.

algorithm for stochastic sensor selection minimizing the expected error covariance, based on Kalman filters with an underlying Hidden Markov Model, is proposed in [73]; it relies on the assumptions of process model linearity and sensor noise Gaussianity.

Finally, another domain where sensor selection techniques are of interest is that of Wireless Sensor Networks (WSN). WSN are networks extended over large geographical regions that feature low power sensors interconnected to each other to receive and transmit data. In [185], an event-based sensor data scheduler deriving an approximate minimum mean-squared error (MMSE) estimator has been developed for remote state estimation over a network. In [123], the sensor selection problem over WSN has been addressed by using Kalman filters with various different cost functions and network constraints and assuming a predetermined time horizon. Finally, a distributed sensor node-level energy management approach for minimizing energy usage has been outlined in [76], which is based on target trajectory prediction through Kalman filters and Interacting Multiple Model (IMM) filters.

In contrast to all the above solutions, continuing with this chapter, we propose a couple of *black-box* sensor selection approaches, applicable to a generic indoor envi-

Figure 3.2: One of the Raspberry Pi Zero boards used in the study.

ronment, and well-suited also for non-linear process models and time series consisting of both real and discrete values. Although we focus on the temperature estimation scenario, the approach can, in principle, be applied to any other prediction task.

Before delving into the heart of the matter, the next subsection will introduce the first application domain under consideration for this study.

3.1.1 The application domain

The first considered scenario is an open space office at Silicon Austria Labs, in Villach (Austria). As shown in Figure 3.1, the room is fairly large, having an overall surface of $127m^2$, and it is characterized by the presence of an always-on air conditioning system, intermittently used workplaces equipped with high-performance workstations, individually controlled radiators, and some windows, at the top and the bottom of the map, that can be independently opened and blinded. Twelve Raspberry Pi Zero boards (Figure 3.2) are deployed for the measurements, each one equipped with an Enviro pHAT sensor board featuring temperature, pressure, light, color, motion, and analog sensors. The recordings are transmitted via WLAN, through a FritzBox access point, towards a Raspberry Pi 3 which acts as a database server. The clients do not store any data: all measured values are sent immediately to the database with a preset periodicity of approximately 10 seconds. Both servers and clients run on Raspbian OS, while the server-side database is based on MySQL. Each client is programmed through a Python 3 script.

The placement of sensors has been organized with the goal of monitoring a variety of operating conditions as large as possible. As an example, sensor 9 is placed near the center of the room and thus ought to be less affected by weather conditions than, for instance, sensors 2, 5, 6, 8, 10 and 11, that are located close to a window. A grid has been superimposed over the map, that allows us to assign to each sensor a unique coordinate, expressed by a tuple (X, Y), with $0 \leq X \leq 10$ and $0 \leq Y \leq 9$. The size of each grid cell is approximately 1.15 m^2.

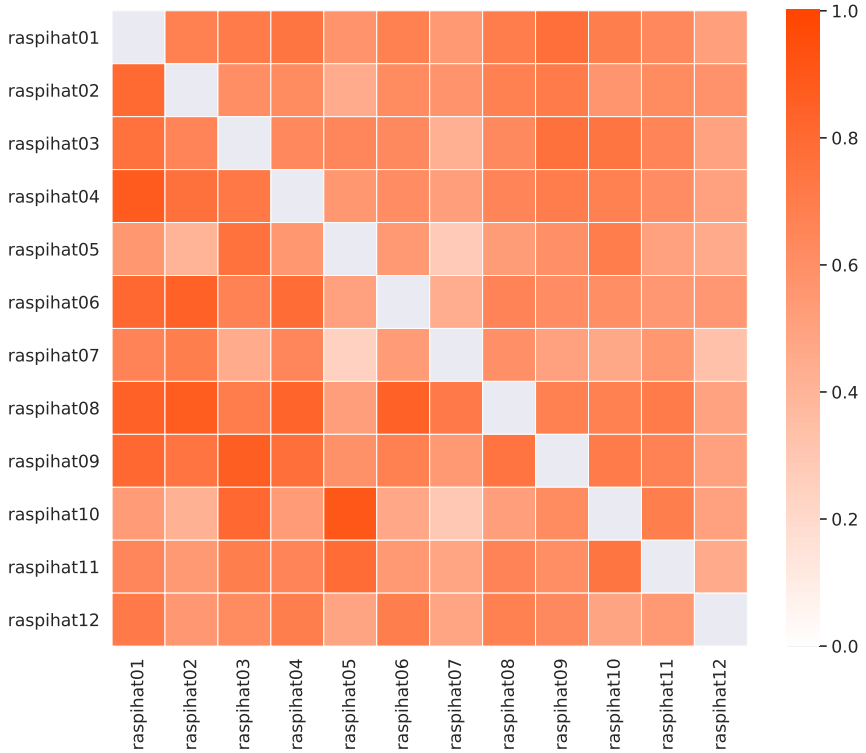


Figure 3.3: Pearson (lower triangular part) and Kendall (upper triangular part) correlation values among the recorded temperatures.

The considered dataset consists of measurements taken from the twelve sensors over the period from 10/23/2019 to 03/03/2020 (henceforth, dates will be represented in the Month/Day/Year format). Data is recorded every 10 seconds, for a total of around 13,500,000 instances. The maximum observed temperature (Celsius) was 48.95, while the minimum was 5.36. This confirms the impact of weather: while in the first case the high temperature can be explained by both the sun influence and the temperature of the Raspberry circuitry, in the latter case it is likely to be caused by an open window placed in close proximity to one of the sensors.

Descriptive analysis

The Pearson correlation values among sensor temperatures are depicted in the lower triangular part of Figure 3.3, and they show that some of them are naturally more correlated than others. This is the case, for instance, with sensors located close to a window in the upper part of the room. Moreover, sensor 9 shows a high correlation with sensor 1, which is not surprising, as both of them are placed near the center of the room. However, there are also some notable exceptions. As an example, sensor 10 correlates more with sensor 5 than with sensor 3, despite the fact that it

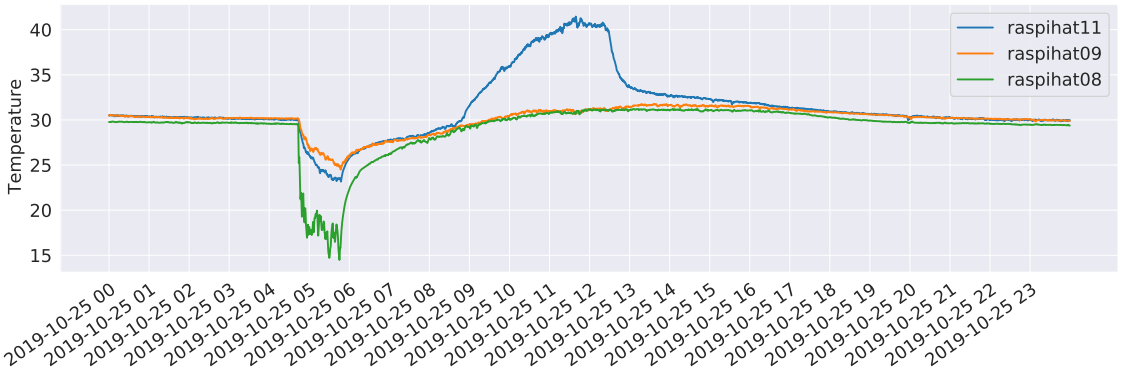


Figure 3.4: Temperatures recorded by sensors 8, 9, and 11 on the day 10/25/2019.

is much closer to the latter than to the former. A similar pattern can be observed for the pair of sensors 12 and 1, as opposed to 12 and 2. These phenomena might be explained by the proximity of a window, the heat emitted by desk lights or other kinds of electrical device, or by the presence of some obstacles that may interfere with air flows, or block the light reaching a specific sensor.

In order to investigate whether these pairs of neighboring sensors share some non-linear relationships or are simply measuring different phenomena, a further analysis based on Kendall's tau, a non-linear rank correlation measure, has been carried out. Results are depicted in the upper triangular part of Figure 3.3. In this case, the correlation between the pair of sensors 12 and 2 is higher than the one estimated for the pair of sensors 12 and 1. Conversely, sensor 10 still correlates more with sensor 5 than with sensor 3, which, not being directly adjacent to a window, is probably measuring a different kind of phenomenon.

As for the temporal evolution of the temperatures, Figure 3.4 shows the measurements of three different sensors for the day 10/25/2019, which follow a typical pattern along a 24 hours period. It is worth noticing that the temperature recorded by sensor 11 has a spike around 11:30 in the morning. This is probably due to the heating effect of direct sunlight, that is not present on sensor 8, which is close to another window but on the opposite side of the building. Instead, the latter sensor shows a drop in the temperature around 5:00 a.m., which can be explained by the daily cleaning staff operations, that include opening the nearby windows to circulate air. Finally, sensor 9 is characterized by a rather stable behavior, being placed close to the center of the room. It is worth pointing out that, as witnessed by the high average temperatures, sensors have not been calibrated. This was done on purpose, in order to evaluate the performance of the proposed methods in a more challenging scenario, also considering that a proper calibration process might not always be possible in a real-world deployment.

As we shall see, in the remainder of the section we will always consider an 80%-20% training-test split of the dataset, randomly assigning entire weeks to the two partitions. The use of this fixed random split in our study can be justified based on several factors. Firstly, such a choice has been made taking into account the fact that the recordings span less than one year and cover months that typically exhibit considerable meteorological variations. Thus, it would have been otherwise difficult for the machine learning approaches that we are going to consider to learn models over a subset of the first days capable of generalizing well to the remaining time period. In addition, upon data inspection, it emerged that even contiguous weeks tend to have rather different behaviors. Secondly, the dataset employed in the study is of a substantial size, ensuring an adequate representation of the data for training, validation, and testing. Additionally, implementing cross-validation techniques could be computationally intensive, requiring significant resources and time. Considering the practical constraints and limitations, such as limited computational capacity, the use of a fixed random split provides a reasonable compromise between computational efficiency and reliable performance assessment.

The performed analysis allows us to conclude that the considered setting is not trivial, as there are some irregularities and local phenomena that influence and differentiate the temperature recordings of the different sensors, making the temperature prediction task quite challenging.

3.1.2 Data pre-processing

In order to carry out the experiments, it is necessary to define a set of features that can be used as predictors by the machine and deep leaning models we are going to develop. To this end, for each temperature measurement, we considered a set of temporal attributes, that are useful to locate the observation in time. They are *sec_from_midnight*, that tracks the number of seconds elapsed from 00:00:00 hour, *dow*, a numerical identifier of the day of the week, and *moy*, a numerical identifier of the month of the year. In order to account for time periodicity (and be able to consider, e.g., the fact that 11:59:59 PM is close to 00:00:00 AM), we encoded each feature by means of two trigonometric transformations:

$$\sin(2 * \pi * x/\delta) \;\; and \;\; \cos(2 * \pi * x/\delta),$$

where x represents the original attribute value and δ is the length of the period, e.g., 12 for the attribute *moy*. As a result, 6 features were obtained: *dow_sin* and *dow_cos*, whose values are shown in Figure 3.5, *moy_sin* and *moy_cos*, whose values are shown in Figure 3.6, and and $sec\_from\_midnight\_sin$ and $sec\_from\_midnight\_cos$, whose values are shown in Figure 3.7.

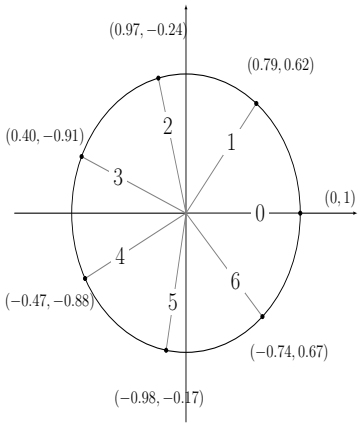


Figure 3.5: Values assigned to (*dow_sin*, *dow_cos*) after a trigonometric transformation of the *dow* feature, where *dow* ranges from 0 (Monday) to 6 (Sunday).

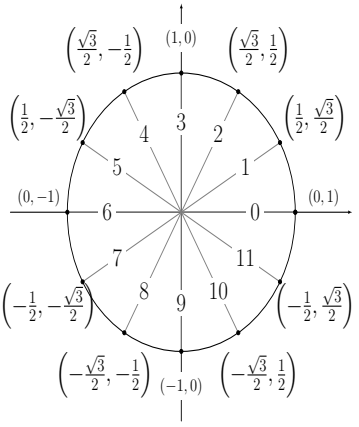


Figure 3.6: Values assigned to (*moy_sin*, *moy_cos*) after a trigonometric transformation of the *moy* feature, where *moy* ranges from 0 (January) to 11 (December).

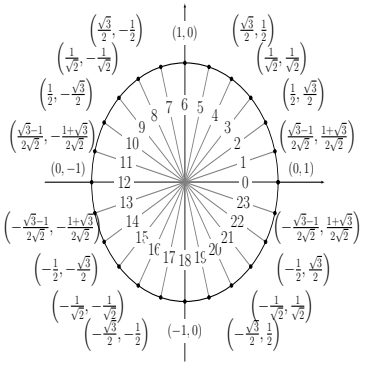


Figure 3.7: Values assigned to (*sec_from_midnight_sin*, *sec_from_midnight_cos*) after a trigonometric transformation of the *sec_from_mid-night* feature ranging from 0 to 86399.

Distance metrics

In this part of the section, the main goal of our work is to study a solution capable of exploiting a subset of sensors to make predictions in place of a sensor to be virtualized. To achieve it, distance metrics can be useful to decide which physical sensors to consider as predictors, and to compute the predicted temperature. Furthermore, distance metrics may be quite useful for the task of optimizing the positioning of a given set of sensors within a certain environment.

The following distance metrics between two sensors $sensor_i$ and $sensor_j$, whose spatial positions are respectively $p_i := (x_i, y_i)$ and $p_j := (x_j, y_j)$, have been considered:

- **Euclidean distance**: length of a line segment between the two points p_i, p_j defined as the L_2-norm $\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$.
- **Manhattan distance**: the L_1-norm of the distance, defined as $|(x_i - x_j)| + |(y_i - y_j)|$.
- **Chebyshev distance**: the L_∞-norm of the distance, defined as $\max\{|(x_i - x_j)|, |(y_i - y_j)|\}$.

- **Genetic Programming distance**: a combination of the previous 3 distances obtained by means of a genetic programming algorithm which generates a computation tree whose leaves may contain the 3 aforementioned distance values or a randomly generated constant and whose internal nodes are the scalar/vector operations defined as a set of primitives.

- **Pearson correlation**: it expresses a possible linear relationship between the statistical variables given by the temperature values of the two sensors, and it is defined as

$$\left(\sum_{k=1}^{n}\left(t_i^k-\bar{t_i}\right)\left(t_j^k-\bar{t_j}\right)\right) / \left(\sqrt{\sum_{k=1}^{n}(t_i^k-\bar{t_i})^2}\sqrt{\sum_{k=1}^{n}(t_j^k-\bar{t_j})^2}\right),$$

  where n is the sample size, t_i^k and t_j^k are the individual sample points for $sensor_i$ and $sensor_j$, that is, the temperature values, and $\bar{t_i}$ and $\bar{t_j}$ are the sample *means*.

- **Kendall correlation**: it expresses a possible ordinal (non-linear) association between the statistical variables given by the temperature values of the two sensors, and it is defined as

$$\frac{2}{n(n-1)}\sum_{k<l} sgn(t_i^k-t_i^l)\ sgn(t_j^k-t_j^l)$$

  where n is the sample size, t_i^k and t_i^l are the individual sample points for $sensor_i$ in position k and l, t_j^k and t_j^l are the individual sample points for $sensor_j$ in position k and l, and $sgn(x)$ is the function $sgn : \mathbb{R} \to \{-1, 0, 1\}$ that returns -1, if $x < 0$, 1, if $x > 0$, and 0 otherwise.

- **SHAP distance**: SHAP is a game-theoretic method that allows one to evaluate the contributions to the final result of the different predictors used in a machine learning model, the relevance of the contribution of a predictor to the model being proportional to its SHAP value [112]. In our case, SHAP values for a generic sensor $sensor_i$ are obtained from an XGBoostRegressor model [41] that predicts the temperature value of $sensor_i$ on the basis of the temperature values of the other sensors. It is worth noticing that such a metric is not symmetric.

- **SAX-CBD distance**: SAX-CBD is a Compression-Based Dissimilarity measure [90] based on the assumption that the size of the compressed file of the concatenation of two discrete time series is inversely proportional to the number of patterns that they share. As a preliminary step, the temperature values

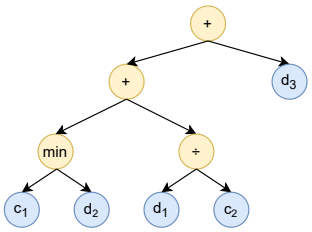
obtained from $sensor_i$ and $sensor_j$ are discretized by means of Symbolic Aggregate approXimation (SAX) [108]; then, the value of the distance is computed as

$$size\_of(compress([d_i \| d_j]))/(size\_of(compress(d_i)) + size\_of(compress(d_j))),$$

where $\|$ is the concatenation operator, d_i and d_j are the time series consisting of, respectively, the discrete temperature values related to $sensor_i$ and $sensor_j$, and $compress(data)$ is the output of the application of the algorithm DEFLATE [140] to *data*.

The estimation of the above metrics is straightforward and it has been made considering only the training set data, in order to obtain for each sensor an ordering of the remaining 11 sensors, from the closest to the farthest one. These ranks are calculated for each distance metric listed above.

Turning to the genetic programming algorithm, it was designed relying on the Distributed Evolutionary Algorithms in Python (DEAP) framework [61]. The hyperparameters used by the EA have been established through grid search tuning performed over a further 90%-10% training-validation subsplit of the training data, randomly assigning entire weeks to the two sets. They are as follows: *population size* = 600 individuals, i.e., computation tree-encoded functions (tested values $[500, 600, 700, 800, 900, 1000]$); *crossover probability* = 0.7 (tested values $[0.5, 0.6, 0.7, 0.8]$); *mutation probability* = 0.4 (tested values $[0.3, 0.4, 0.5, 0.6]$); *max generations* = 100 (tested values $[50, 100, 250, 500]$). As for the evolutionary operators, we employed one point crossover and a mutation where a randomly chosen primitive from an individual is replaced by another randomly chosen operation within the primitive set. To avoid bloat, that is, an excessive increase in mean program size without a corresponding improvement in fitness, we placed a static limit of 17 on the children's height (DEAP's staticLimit), as suggested by Koza in [98]. The chosen selection method is the double tournament [111], which evaluates both the fitness and the size of the individuals in order to discriminate good solutions, following a 3-individuals fitness-based first tournament and a size-based second tournament with a parsimony size of 1.4 (tested values $[1.2, 1.4, 1.6, 1.8]$). This last tournament favours the choice of low-complexity solutions, represented by trees of limited height. Individuals are built considering as terminal leaves the three (0-1 normalized) distance metrics *Euclidean distance, Manhattan distance*, and *Chebyshev distance*, and a set of random constants ranging from -1 to 1. The set of primitives consists of the following scalar/vector operations: $\min, \max, +, -, *, \div, \log_{10}$, *exponentiation*, *square_root*, *negation*, and *absolute_value*. Note that, despite the normalization step performed on the distance metrics, the *absolute_value* operation is still useful given the presence of potentially negative constants in the tree.



Node	Value
d_1	Euclidean distance
d_2	Manhattan distance
d_3	Chebyshev distance
c_1	$\simeq$ 0.296
c_2	$\simeq$ 0.480

Figure 3.8: The computation tree generated by the genetic programming algorithm representing a spatial distance metric.

In order to determine the fitness function, we considered a the aforementioned 90%-10% training-validation subsplit of the training data. In both datasets, each instance consists of a label, that is, the temperature recording of a given sensor, and a list of predictors, that are, the 11 temperature values of the other sensors, and the temporal features *sec_from_midnight_sin*, *sec_from_midnight_cos*, *dow_sin*, *dow_cos*, *moy_sin*, and *moy_cos*. The fitness function was computed for each individual as follows: we assessed the prediction error obtained from a series of linear regression models built on the training split and evaluated on the validation split; different models were trained and evaluated, considering each different sensor as a target and increasingly discarding other predictor sensors according to the rank defined by the distance function encoded by the individual, starting from the sensor with the highest value. Then, for each number of considered predictors, we summed the resulting prediction errors, coming from the different target sensors, obtaining an error curve. Finally, to determine the fitness value, we calculated the area under the curve. The computation tree generated by the genetic programming algorithm is shown in Figure 3.8. It is equivalent to the function $GP\_function(d_1, d_2, d_3) = \min\{c_1, d_2\} + d_1/c_2 + d_3$, where d_1 is the *Euclidean distance*, d_2 is the *Manhattan distance*, d_3 is the *Chebyshev distance*, $c_1 \simeq 0.296$, and $c_2 \simeq 0.480$. It is clear that the variables d_1, d_2, and d_3 generally provide an incrementally increasing contribution in this function, but the precise nature and rate of increase depend on their specific values and relationships with the constants c_1 and c_2.

The evaluation of the rankings generated by each distance metric d was carried out according to the following procedure on the 80%-20% training-test split:

- for each sensor $sensor_i$, the rank $rank_{d,i}$ of the other sensors according to the metric d was considered;
- then, we proceeded in an iterative way: for $k \in \{0, \ldots, 10\}$, k sensors among the worst ones in $rank_{d,i}$ were discarded and a regression model was built. In more detail,

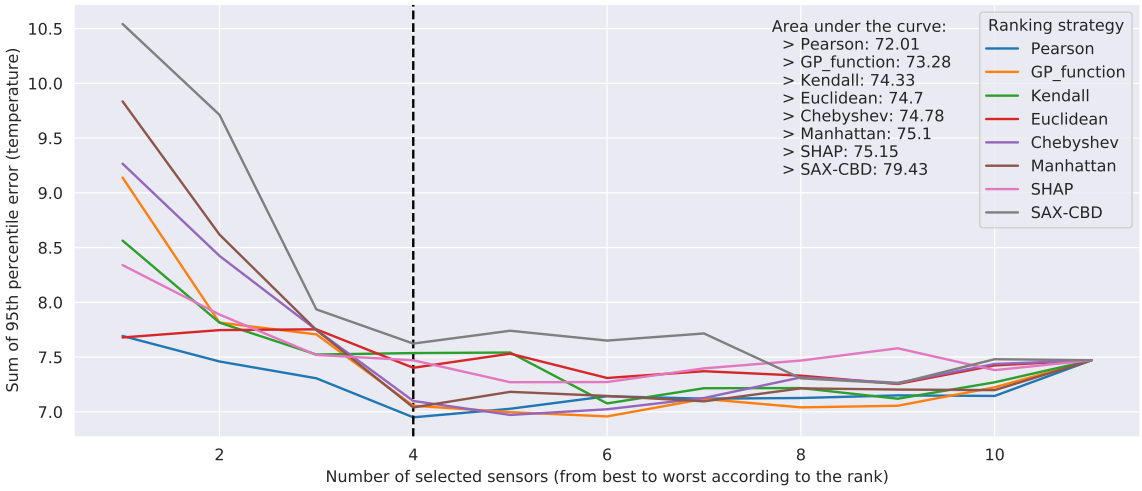


Figure 3.9: Performance of linear regression, evaluated discarding sensors based on training set ranks. The dashed vertical line represents the elbow of the *Pearson* error graph.

- the sensors whose temperature values were to be used as predictors were determined considering the set of all sensors, except $sensor_i$ and the k sensors located in the last k positions of $rank_{d,i}$;
- exploiting the training set data, a linear regression model was built to predict the temperature of $sensor_i$ using as input the temperature values of the sensors selected in the previous step and the features *moy_sin*, *moy_cos*, *dow_sin*, *dow_cos*, *seconds_from_midnight_sin*, and *seconds_ from _midnight_ cos*;
- the resulting model was evaluated on the test set.

For each value of k, the sum of the $95th$ percentile of the absolute errors obtained for each predicted sensor was computed, obtaining a curve over k. Then, the final error for each metric was determined by calculating the area under the considered curve. It is worth noticing that here the $95th$ percentile can be thought of as a worst-case prediction scenario.

The outcome of the evaluation is reported in Figure 3.9. The curves show a bowl-shaped pattern. This can be explained by the fact that the first sensors being discarded may have little correlation with the temperatures to predict, and thus they may interfere with the accuracy of the final result. On the other hand, the lastly discarded ones were probably carrying useful information. The best metrics turned out to be *genetic programming distance* and *Pearson correlation*. More precisely, *genetic programming distance* provided the best results considering a subset of at

least 5 sensors as predictors and, as expected, outperformed in general all the other metrics based exclusively on spatial distances. The *Pearson correlation* showed an overall better performance than *genetic programming distance*, in particular when 4 or less sensors were considered as predictors. However, *Pearson correlation* cannot be employed in a more general setting, that is, to determine the best sensors to use as features to predict the temperature at a given cell, where a reference physical sensor may or may not be present. As for the *genetic programming distance*, while we cannot exclude its ability to generalize on each cell, the latter should be actually demonstrated, and it will be the subject of future investigations based on physical data simulations. As a last remark, we observe that Figure 3.9 clearly shows that excellent results, in terms of prediction error, can be obtained from just a subset of 4–6 sensors, while they begin getting worse when considering 3 or fewer sensors.

3.1.3 Sensors selection

In order to automatically select a subset of sensors to be used as predictors, we need to specify a procedure to determine the number n_{refs} of sensors to select and to establish which sensors to actually consider. To this end, we make use of a procedure based on the Borda count voting method [19]. As a preliminary step, let us introduce some auxiliary notions. First, for $i \in \{1, \ldots, 12\}$, let $rank_i$ be the ranking of sensor i obtained by sorting in descending order the remaining 11 sensors according to their Pearson correlation with reference to sensor i. Then, let us define w_i, with $i \in \{1, \ldots, 12\}$, as the weight of $rank_i$, defined as $1 - (\varepsilon_i / \max_{i \in \{1,\ldots,12\}} \varepsilon_i)$, where ε_i is given by the sum of the 95th percentile of the absolute errors evaluated for $rank_i$, computed training different LinearRegression models varying the number of sensors used as predictors from 1 to 11, as described for the case of the metric evaluation procedure at the end of the previous subsection (Section 3.1.2).

The procedure consists of the following five steps: (i) for each sensor i, we compute $rank_i$; (ii) for each sensor j, we determine its weighted Borda count, which is defined as $vote_j = \sum_{i \in \{1,\ldots,12\},\ i \neq j} (n_{sensors} - pos_i(j)) \cdot w_i$, where $pos_i(j)$ is the position of sensor j in $rank_i$, $n_{sensors} = 12$ is the total number of used sensors, and w_i is the weight of $rank_i$; (iii) the sensors are sorted in descending order according to their final weighted Borda count vote; (iv) an approximation of the elbow of the curve obtained in the previous step is computed by using the Kneedle algorithm [165] – the x-axis value corresponding to the elbow represents the point of maximum curvature of the graph, and the best trade-off between prediction accuracy and number of sensors, after which a law of diminishing returns applies: we choose it to be the n_{refs} value which, in our case, corresponds to 4 reference sensors (dashed line in Figure 3.10); (v) finally, the first n_{refs} sensors are selected as the reference ones.

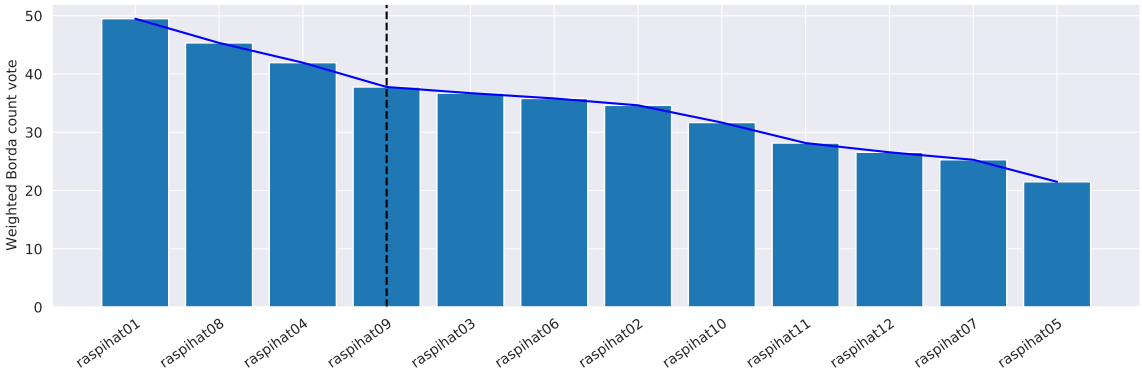


Figure 3.10: Weighted Borda count vote for each sensor. The vertical line represents the elbow of the graph, and it separates the selected sensors from the discarded ones.

It is worth to underline that the weights w_i in the vote counting formula allow us to give greater importance to the ranks that provided better results with respect to the error obtained on the validation data split. In addition, the sensors located on the left side of elbow intuitively correspond, by construction, to the best predictors.

Figure 3.10 shows the weighted Borda count votes for all sensors obtained by applying the procedure to the considered setting. The proposed criterion led to the selection of sensors $1, 8, 4$, and 9 as the references for the prediction. It is noteworthy that the elbow estimate also corresponds to the minimum error point of *Pearson distance* in Figure 3.9. Even if in our case no ex aequo placements occur, as a methodology to deal with them, we suggest to consider for each sensor the median of its Pearson correlations with respect to the other ones, prioritizing those with higher values.

3.1.4 Feature selection

When performing tasks, such as virtual sensing of temperature, the selection of features used to train the considered models is crucial to achieving high prediction accuracy. The inclusion of irrelevant or redundant features can lead to overfitting, increased model complexity, and computational costs. Therefore, feature selection is an essential step to improve the model's performance and generalization ability.

On the basis of the feature engineering and sensor selection phases, we identified 16 attributes that describe each observation: the temporal features *sec_from_midnight_sin*, *sec_from_midnight_cos*, *dow_sin*, *dow_cos*, *moy_sin*, *moy_cos*, the spatial features *01_ref_dist*, *04_ref_dist*, *08_ref_dist*, *09_ref_dist*, and the reference temperatures *01_ref_temp*, *04_ref_temp*, *08_ref_temp*, *09_ref_temp*. To them, we added *X_coord* and *Y_coord*, that is, the two grid coordinates of the sensor that recorded the observation.

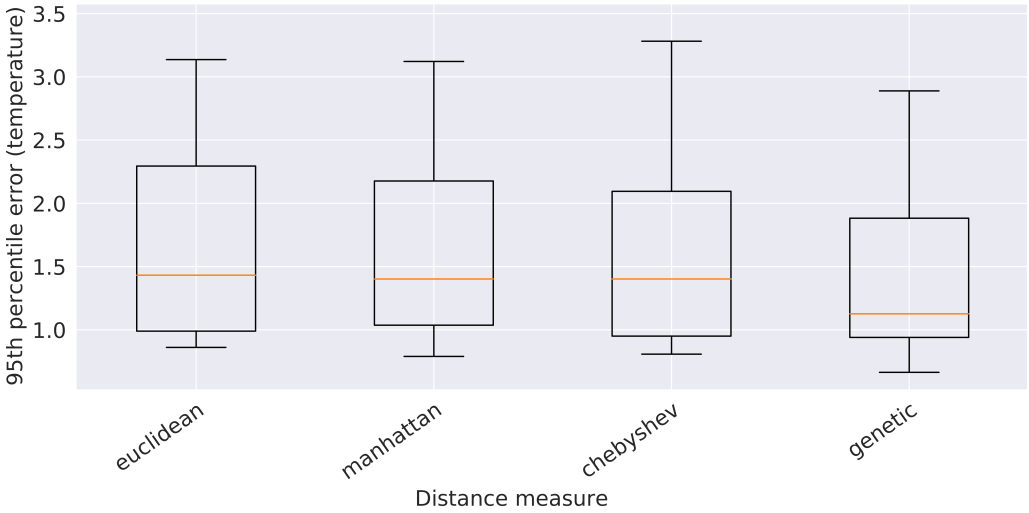


Figure 3.11: Boxplots of the 95th percentile of the error provided by the XGBoost models built on the 5 spatial distances considered in this study.

At this point, it is necessary to establish which spatial distance metric to consider among *Euclidean distance*, *Manhattan distance*, *Chebyshev distance*, and their combination obtained via *genetic programming* for the estimate of *01_ref_dist*, *04_ref_dist*, *08_ref_dist*, and *09_ref_dist*. To this end, we determined the 95th percentile of the validation error values obtained from 4 global 16-attribute XGBoostRegressor models, one for each spatial distance metric, trained on the usual 90% subsplit of the original training data pertaining to all the sensors, except for the reference ones (that are already used as predictors). The outcome of such an analysis is depicted in Figure 3.11 and led to the selection of the *genetic programming distance*, that outperformed all the other ones. At the end, the following 16 attributes have been chosen to describe each observation: the 6 temporal features, the 4 reference temperatures, the 4 spatial features *01_ref_gpdist*, *04_ref_gpdist*, *08_ref_gpdist*, and *09_ref_gpdist*, which are the genetic programming distances from the reference sensors, and the 2 grid coordinates *X_coord* and *Y_coord* of the sensor that recorded the observation.

Since some of these attributes may be redundant, we executed a 2-step feature selection process working on the training split. As a preliminary data preparation step, all attributes were standardized by subtracting their mean and dividing by their standard deviation. The first selection step searched for highly correlated attributes, that is, attributes with a Pearson correlation value above 99%. As a matter of fact, no feature was removed from the dataset by this step. The second step evaluated the potential impact of the remaining attributes on the final prediction. To this end, the SHAP values extracted from a single global XGBoostRegressor model trained on the training data split related to all the sensors, except for the reference ones, were

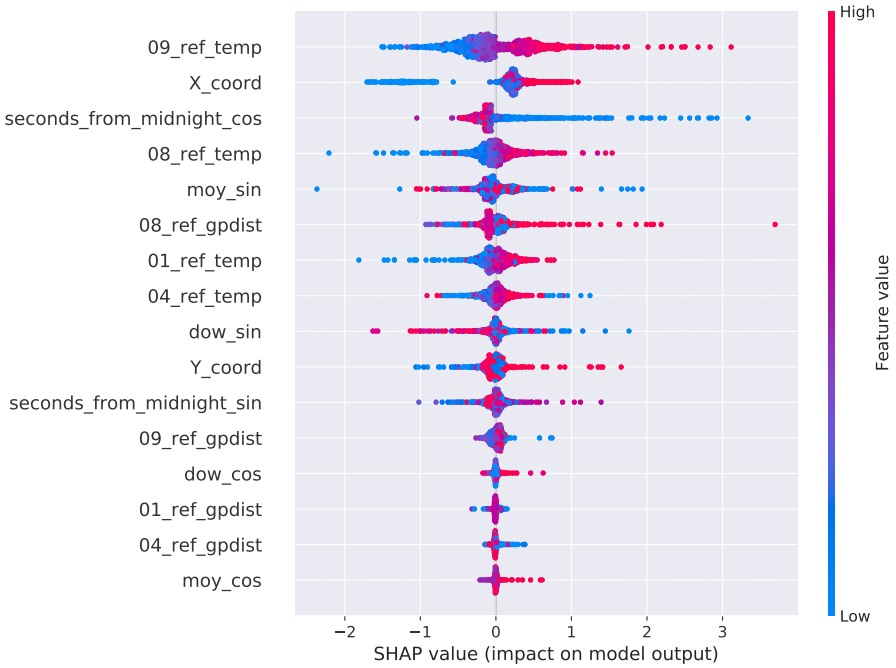


Figure 3.12: SHAP values of the attributes considered in the second step of the feature selection process.

taken into consideration. The outcome of such an analysis is depicted in Figure 3.12. We first observe that all reference temperature values have a large impact on the final prediction and, naturally, higher values of these attributes increase the value of the prediction. Focusing on the office map, there is a clear variation across the X and Y axes. As for the Y axis, from the distribution of the values, we can conclude that, during the data collection period, the northern side of the room was generally warmer than the southern one. As for the X axis, the eastern side of the room seems to be warmer than the western one. Interestingly, the *moy_cos* and *dow_cos* features do not seem to be important for the overall prediction when compared with the counterpart obtained from the sine transformation. As pointed out by Figure 3.5 and Figure 3.6, this means that the contribution to the prediction given by the features that discriminate the first half of the week/year from the second half are more important than those that discriminate the first/fourth quarters from the second/third ones. Furthermore, the genetic programming distances from sensors 1, 4, and 9 are considered of marginal importance when compared to the distance from sensor 8. On the basis of the SHAP results, we ultimately decided to remove the 5 attributes *dow_cos*, *moy_cos*, *01_ref_gpdist*, *04_ref_gpdist*, and *09_ref_gpdist*, ending up with a total of 11 ones.

From a general point of view, the first correlation-based feature selection step

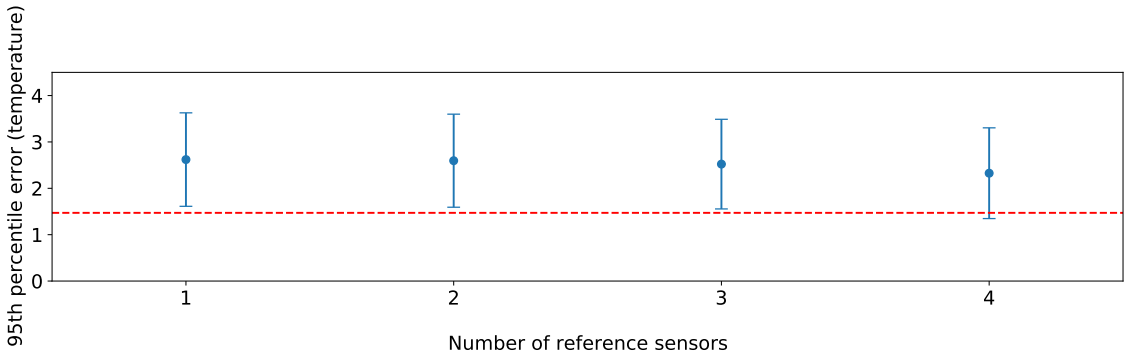


Figure 3.13: Results obtained from XGBoost on all possible combinations of k reference sensors, with $k \in \{1, \ldots, 4\}$. For each value of k, the vertical line represents the extent of the errors given by the different combinations, while the dots represent the median error. The red dashed horizontal line represents the error obtained by the subset of reference sensors selected by our approach.

must be considered as a preliminary, coarse screening of the predictor variables; it should discard a feature when it is found to be almost identical to another one in the dataset, without the risk of removing predictors that might still be preserved by the subsequent, SHAP-based feature selection step. This is the reason why we recommend to rely on a high threshold, which should nevertheless be established considering the specific scenario. Indeed, such a pre-screening may be useful in a more general situation, characterized by a large amount of input features, to reduce the time requirements needed to train the XGBoost model in the second step of the feature selection phase. In our case, as already discussed, relying on a correlation threshold of 99% led to no attribute being discarded. However, lowering the threshold to 94% would have led to the removal of feature *04_ref_gpdist* which would have been also discarded in the subsequent SHAP-based step. To remove the first feature kept in the SHAP-based step, namely *04_ref_temp*, the correlation threshold should have been reduced to a value less than or equal to 88%.

3.1.5 Comparison with a brute force approach

As a final experiment, in order to evaluate the effectiveness of the proposed sensor selection procedure, we carried out a comparison with a brute force approach which considers any possible combination of 4 or less sensors chosen as the reference ones. For each possible combination, we trained an XGBoost model on the training data split based on the features discussed in Section 3.1.4, before the feature selection phase. To ensure comparability, we performed our evaluation considering the test

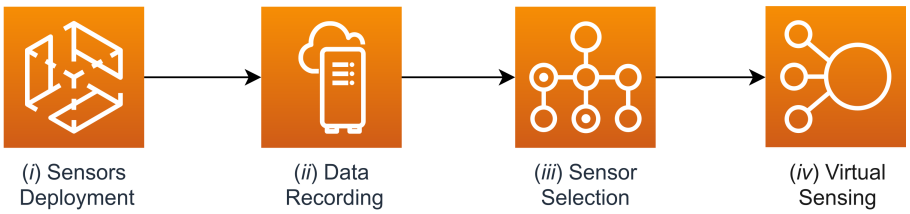


Figure 3.14: High-level steps which characterize the proposed procedure.

data split of the four sensors 2, 5, 7, and 10, since they cover the whole room and are also typically placed in the lower tier of the rankings analysed in Section 3.1.2. Excluding the aforementioned 4 sensors, the possible reference sensor combinations are $\sum_{k\in\{1,\dots,4\}} \binom{8}{k}$. As for the cardinality k of the subset of selected reference sensors, a maximum value of 4 was chosen to ensure that there is no combination with a lower number of sensors capable of providing a better prediction accuracy than the one obtained with the proposed solution. Results are shown in Figure 3.13. The performance of the proposed solution is very close to the optimal one achieved by brute force. This is remarkable, especially considering that the brute force solution performs an exhaustive search in the problem space, and can thus be applied only in scenarios involving a small number of sensors.

3.1.6 Discussion

While it is clear that prediction performances are expected to improve together with the number of sensors and historical training data available, it is typically worth to find a trade-off between the accuracy of the models and the cost of the employed resources.

Here, we provide a final overview of the overall procedure for this first proposed sensor selection solution detailed thus far in this chapter, whose steps, as portrayed in Figure 3.14, are as follows: (*i*) a set of sensors is placed inside a room; (*ii*) the temperature values measured by these sensors are collected over a significant period of time; (*iii*) following the steps outlined in Section 3.1.3, a subset of sensors is selected from those present in the room by exploiting the Pearson correlation of their temperature measurements; (*iv*) the other sensors can be removed from the room and replaced by the output of predictive models built using information from the remaining reference sensors, as we shall see in Section 4.1.1.

Specifically, it is once again worth to highlight the role played by the sensor selection technique, which, in the considered scenario, provided a very good result when compared to a plain brute force approach. This could be achieved leveraging the weighted Borda count method in combination with the Kneedle algorithm: the

former allows us to give a greater importance to the sensor ranks that provided the better results, thus favoring sensors that ought to be the most relevant predictors; the latter reasonably approximates a trade-off point at which the cost of increasing the number of sensors used as predictors is no longer worth the corresponding performance benefit.

3.2 Sensor placement in simulated indoor environments

With the aim of proposing and evaluating techniques capable of monitoring temperatures in uncovered or hard-to-reach locations, we design and produced a set of simulation data by means of computational fluid dynamics algorithms for a generic room model, concerning various environmental and usage conditions, as well as differing internal object arrangements. Using this type of simulation data has a number of advantages. Physical simulations provide a controlled and replicable environment, allowing for the generation of large quantities of high-quality data with known ground truth values. This is particularly important when developing and testing smart sensing solutions, as it allows for more accurate and reliable training and validation of the models. In contrast, real-world sensor data can be noisy and subject to various environmental factors, making it difficult to discern the true underlying patterns and trends. Secondly, physical simulations can be used to model a wide range of scenarios and conditions that may not be feasible or practical to reproduce in the real world. Additionally, simulations can be scaled up or down to generate data at different resolutions and spatial scales, making it possible to tailor the datasets to the specific needs of the smart sensing application.

Before describing the devised sensor placement solution in Section 3.2.2, in the following part we introduce the second application domain under consideration for this study.

3.2.1 The application domain

The considered scenario for the physical simulation data is a generic indoor environment inspired by an actual room located at the Silicon Austria Labs office in Villach (Austria). As shown in Figure 3.15, the room has an overall surface of 22 m^2 and a height of $3,1$ m. In order to simulate a common usage of the room the following objects have been inserted into it mimicking different configurations: convective heaters, representing a heat source with an air flow; non-convective heaters, such as stoves or radiators, representing sources of heat diffused only by radiation; and,

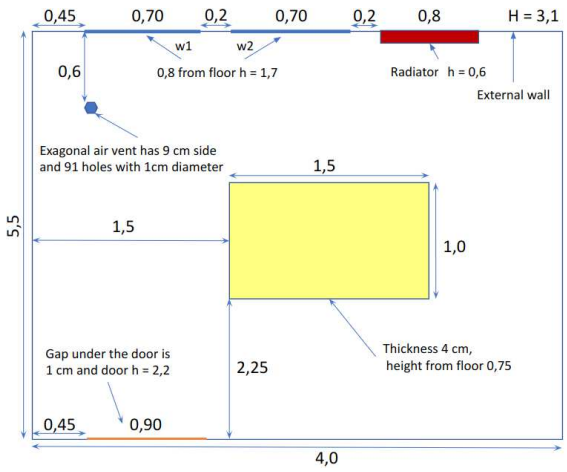


Figure 3.15: Original configuration of the modeled room. Measures are expressed in meters if not differently specified.

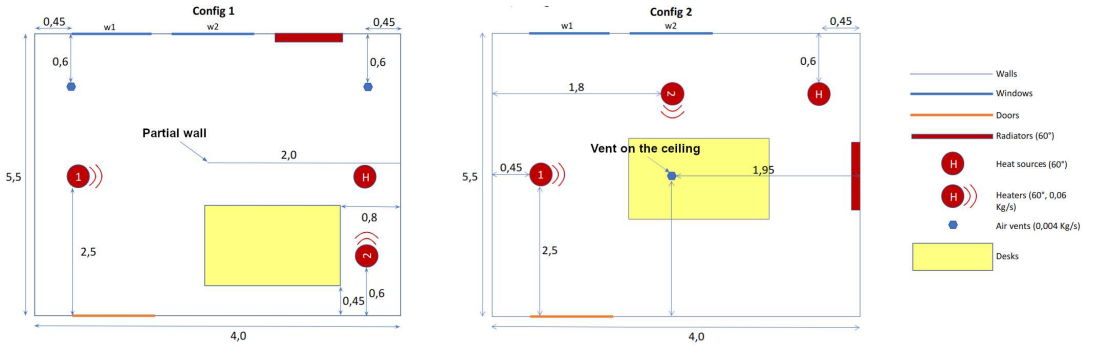


Figure 3.16: Configurations of the modeled indoor room considered for the physical simulations. Measures are expressed in meters.

partial walls and wooden tables that can affect the air distribution. Room walls are conductive and have windows and doors that can be individually opened and closed; in our modeled scenario one of the walls is external, while the others are internal to the building. In addition, there are air vents on the floor and ceiling that are part of the air conditioning system.

As shown in Figure 3.16, we considered 2 different room configurations in order to evaluate the generalization capability of the approaches proposed in this study. Moreover, for each of these configurations two conditions have been taken into account. The first, winter-related, is characterized by a temperature of 19°C indoors and 5°C outdoors, while the second, summer-related, by 25°C indoors and 35°C outdoors. The simulated events are of various types: opening/closing of windows and doors, switching on/off of heaters, radiators and air conditioning systems. The

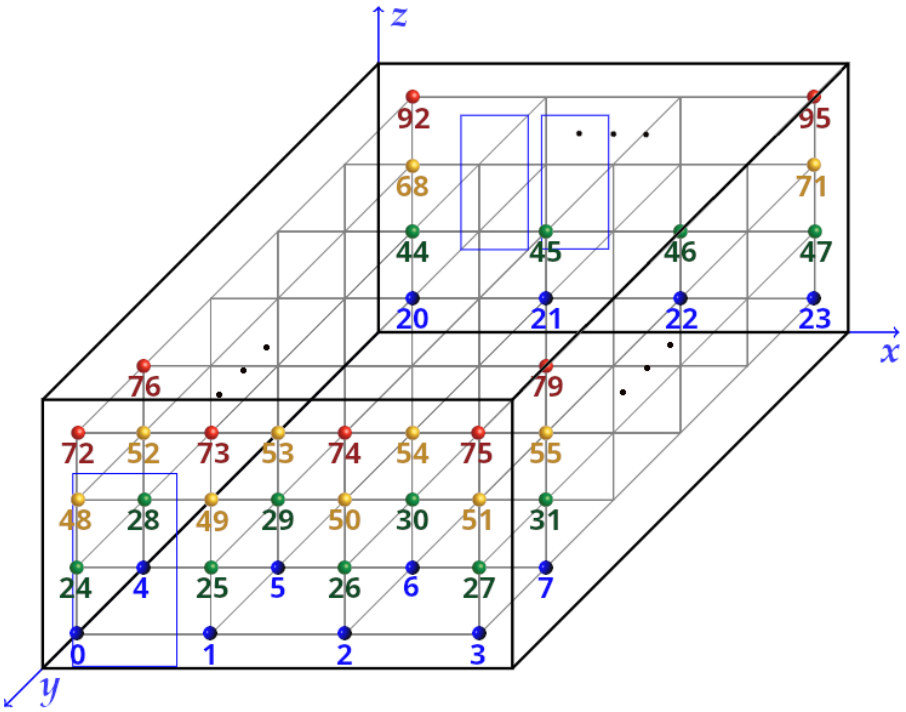


Figure 3.17: Room sensor grid map.

simulations of these events are carried out relying on CFD algorithms for transient conditions and take place both individually and in a sequence until a temperature stabilization point is reached. The sequence of simulated events for the winter season, hereinafter referred to as Seq_w, is as follows: (1) temperatures stabilized (19°C inside, 5°C outside); (2) radiator on (60°C); (3) heat source H on (60°C); (4) air conditioning vents on (22°C, 0.004 Kg/s); (5) window $w2$ opened; (6) door opened; (7) window $w2$ closed and heater 1 on (60°C, 0.06 Kg/s); (8) door closed; (9) heater 2 on (60°C, 0.06 Kg/s); (10) heater 2 reoriented towards west.

Finally, for the summer season the following sequence of events, hereinafter referred to as Seq_s, has been simulated: (1) temperatures stabilized (25°C inside, 35°C outside); (2) heat source H on (60°C); air conditioning vents on (17°C, 0.004 Kg/s); (3) window $w2$ opened; (4) door opened; (5) window $w2$ and door closed; (6) heater 1 on (60°C, 0.06 Kg/s); (7) heater 2 on (60°C, 0.06 Kg/s); (8) heater 2 reoriented towards west.

The ambient temperature is recorded with a 10 seconds frequency through monitors acting as sensor devices arranged inside the simulated environment as vertices of a cubic grid that divides the x, y, z axes of the room into equally separated levels 1 m away from each other. More precisely, in our case the x axis is divided into 4 levels (corresponding to 0.5 m, 1.5 m, 2.5 m and 3.5 m from the western side of

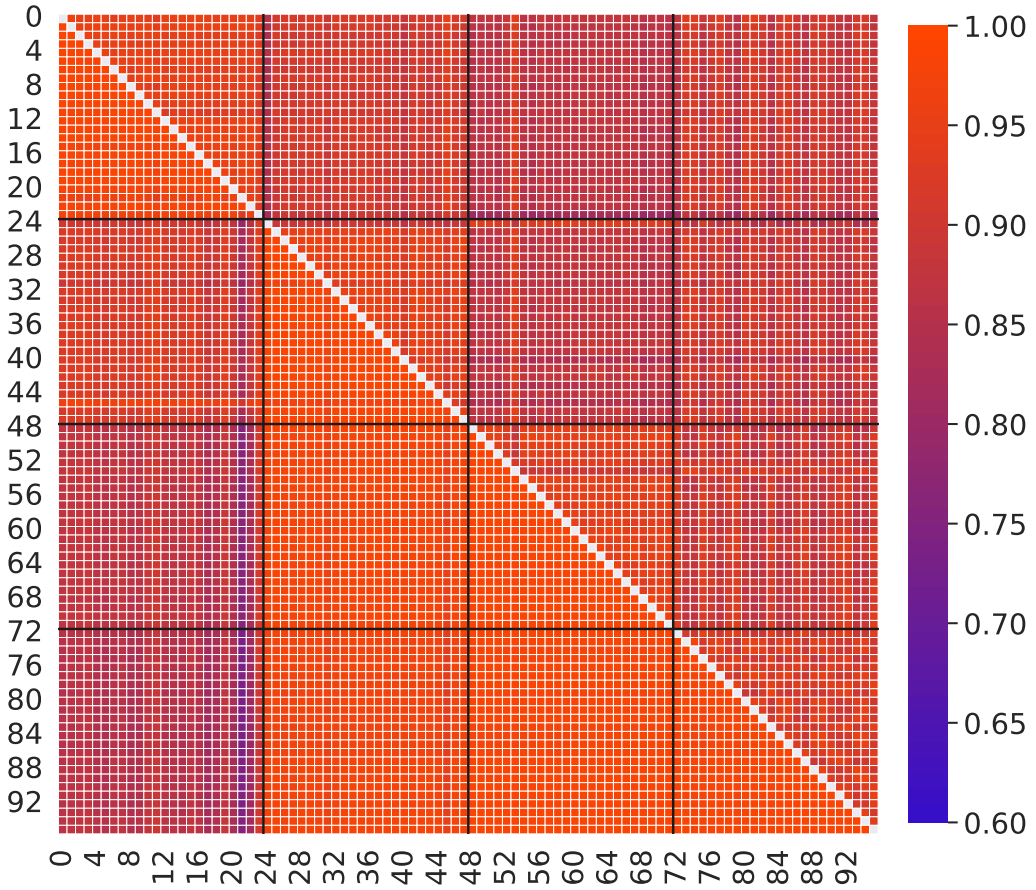


Figure 3.18: Pearson (lower triangular part) and Kendall (upper triangular part) correlation values among the recorded temperatures. The black lines split every z axis level.

the room), the y axis into 6 (corresponding to 0.25 m, 1.25 m, 2.25 m, 3.25 m and 4.25 m from the southern side of the room) and the z axis into 4 (corresponding to 0.05 m, 1.05 m, 2.05 m and 3.05 m from the floor), for a total of 96 temperature sensors. As illustrated in Figure 3.17, the latter are numbered from 0 to 95 in ascending order with respect to the lexicographical ordering of their z, y, x coordinates. Finally, every element inside the room is represented by (x, y, z) coordinates in the 3-dimensional Euclidean space $\mathbb{R}^3$.

Descriptive Analysis

In this part of the section we analyze the salient characteristics of the data obtained by simulating the event sequences Seq_w and Seq_s for room configuration 1. The Pearson correlation values among sensor temperatures are shown in the lower triangular part of Figure 3.18. It can be clearly seen that most of them are highly

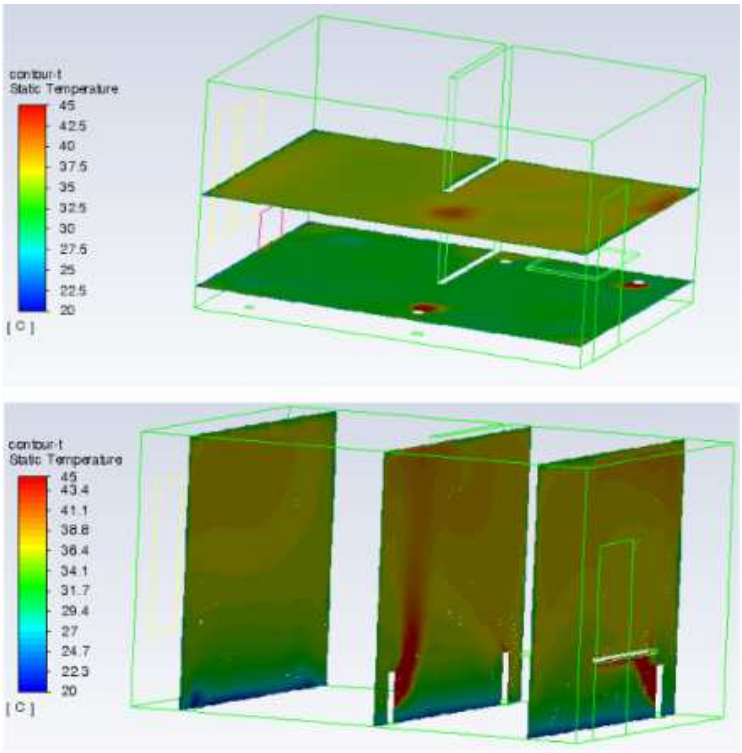


Figure 3.19: Instance of the configuration 1 room thermal mapping with simulated 2D temperature section cuts.

correlated (except for sensors $21, 22$ located, respectively, in front of the radiator and window $w2$ at $0.05m$ above the floor). This means that the information carried by the provided sensors is largely redundant and, therefore, a subset of them is sufficient to accurately monitor the environment. Furthermore, sensors at floor level are relatively poorly correlated with respect to those at other elevation levels In general, it can be seen that correlation values tend to be stratified for sensors located at the same z-axis level.

In order to investigate non-linear relationships between sensors, a further analysis based on Kendall's tau, a non-linear rank correlation measure, has been depicted in the upper triangular part of Figure 3.18. In this case, correlations of sensors at floor level (first 24 rows of the matrix) are still relatively poorly correlated with respect to those at other z-axis levels. However, unlike Pearson correlation there are some sensors which have a high correlation with the ones near the roof on the last z-level (i.e., in the upper right square of the matrix). This means that a high-capacity model is preferable to capture these relationships adequately.

As for the temperatures measured in the simulations, they vary from a minimum of 7.56°C to a maximum of 43.67°C. Figure 3.19 shows an instance of the simulated

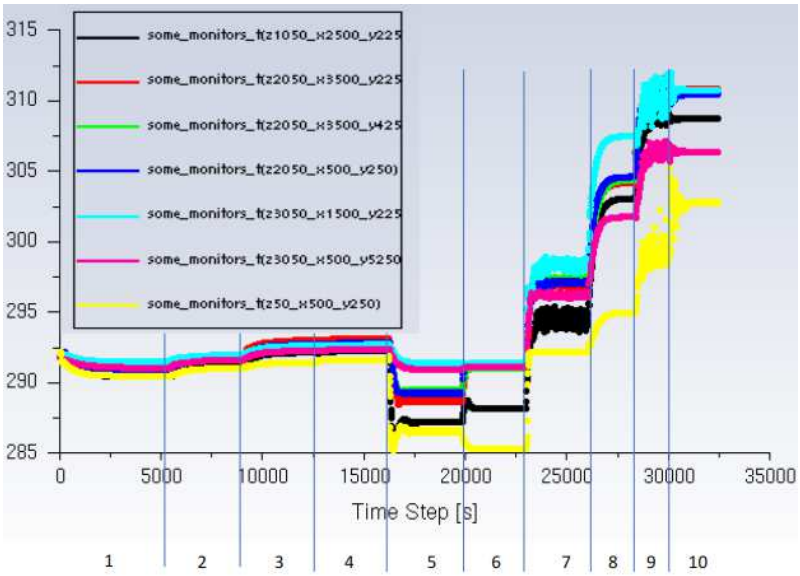


Figure 3.20: Simulated sensors temperature values in Kelvin for sequence Seq_w in room configuration 1 horizontally divided by event number.

3D room with temperature values depicted by means of vertical and horizontal section cuts. The graphs of the observed temperatures related to each of the 10 events which make up the winter sequence Seq_w for configuration 1 are presented in Figure 3.20. As can be seen, there are perturbations in particular during the event 9 in which the flow of the heater 2 overlaps with the one generated by heater 1. This reveals that the considered setting is not trivial, as there are some irregularities and phenomena which influence the temperature recordings at different positions, making the prediction task challenging.

3.2.2 Sensors placement

In this part of the section, a solution to the optimal sensor placement problem based on graph attention networks (GATs) is proposed. In our case, the goal is to provide a methodology able to select a subset of sensors among those present in the grid provided with the simulations. Thereafter, we make a comparison with the approach previously proposed in Section 3.1.3 and a brute-force baseline consisting of random sensor subsets. It is worth noticing that, by leveraging the grid of sensors encompassed in the simulation data, it is possible to reduce the sensor placement problem to that of sensor selection, as previously formulated in Section 3.1.3.

3.2.3 Graph attention networks-based solution

Graph attention networks [176] are a kind of neural network architecture able to operate on graph-structured data leveraging masked self-attention layers in which nodes are able to attend over their neighborhoods with different level of attention. Briefly, in a node embedding learning phase, attention layers allow to focus only on the edges of the graph that are relevant to the task of interest.

More formally, given the directed graph $G = (V, E)$, a set of nodes features $\mathbf{h_v} = \{\mathbf{h}_i | v_i \in V\}, h_i \in \mathbb{R}^{F_v}$ where F_v is the number of features associated to each node and a set of edge features $\mathbf{h_e} = \{\mathbf{h}_{i,j} | (v_i, v_j) \in E\}, \mathbf{h}_{i,j} \in \mathbb{R}^{F_e}$ where F_e is the number of features associated to each edge, the graph attention layer produces a new set of node embeddings $\mathbf{h_v}' = \{\mathbf{h}'_i | v_i \in V\}, h'_i \in \mathbb{R}^{F'_v}$ (i.e., of potentially different cardinality F'_v). The attention coefficient indicating the importance of node v_j's features to node v_i is defined as

$$\alpha_{i,j} = \frac{\exp(\mathbf{a}^T LeakyReLU(\mathbf{W_a}[\mathbf{h}_i \| \mathbf{h}_j \| \mathbf{h}_{j,i}])}{\sum_{k \in \mathcal{N}_i} \exp(\mathbf{a}^T LeakyReLU(\mathbf{W_a}[\mathbf{h}_i \| \mathbf{h}_k \| \mathbf{h}_{k,i}])}$$

where $\mathbf{W_a} \in \mathbb{R}^{F'_v \times (2F_v + F_e)}$ is a learnable linear transformation weight matrix, $\mathbf{a} \in \mathbb{R}^{F'_v}$ is a learnable weight vector, $\mathcal{N}_i$ is the neighborhood of the node v_i, $\|$ is the concatenation operation, $\cdot^T$ represents a transposition and $LeakyReLU$ is the nonlinear leaky rectified linear function with negative input slope of 0.2. The attention coefficients are computed only for pairs of nodes actually connected by an edge $e \in E$ of G. Finally, the new node embedding for the node v_i is computed as $\mathbf{h}'_i = \sum_{v_j \in \mathcal{N}_i} \alpha_{i,j} \mathbf{W} \mathbf{h}_j$, with $\mathbf{W} \in \mathbb{R}^{F'_v \times F_v}$ learnable linear transformation weight matrix.

The proposed sensors selection procedure is designed as follows: (i) for each sensor v_i in V a model GAT_i is trained to forecast the temperature values observed in v_i using the temperatures observed in the remaining sensors $v_j \in V \setminus \{v_i\}$; (ii) then, the rank score $score_j$ for all nodes $v_j \in V$ is computed as

$$score_j = \frac{\sum_{v_i \in V \setminus \{v_j\}} \alpha_{i,j} \; deg_{in}(v_i)}{deg_{out}(v_j)}$$

where $\alpha_{i,j}$ is the attention coefficient extracted from the trained model GAT_i predicting the temperatures of node v_i, $deg_{out}(v_j)$ is a function returning the number of outlinks from a node $v_j \in V$, and $deg_{in}(v_i)$ is a function returning the number of inlinks to a node $v_i \in V$. Intuitively, dividing by $deg_{out}(v_j)$ allows to equalize the contribution of nodes with different number of outgoing edges; while, multiplying by $deg_{in}(v_i)$ allows to limit the softmax distribution of attention coefficients towards nodes with a high number of incoming edges; (iii) the sensors in V are sorted in

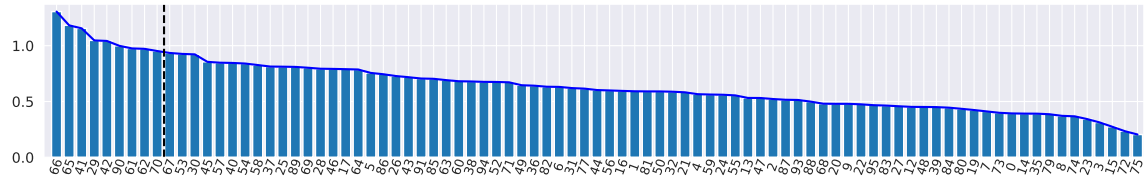


Figure 3.21: Graph attention-based score for the considered train sensors. The vertical line represents the elbow of the graph and separates the selected sensors from the discarded ones.

descending order according to their rank score; (iv) the number of selected sensors n_{refs} is computed by approximating the elbow of the curve obtained in the previous step through the Kneedle algorithm [165] – the x axis value corresponding to the elbow represents the point of maximum curvature of the graph, and the best trade-off between prediction accuracy and number of sensors used – we choose it to be the n_{refs} value; (v) finally, the first n_{refs} sensors are selected as the reference ones.

3.2.4 Experiment setup

The experimentation has been carried out on the Seq_w winter sequence simulation data for room configuration 1 shown in Figure 3.16. The considered graph $G = (V, E)$ is formed by a set of node labels $V = \{0, \ldots, N-1\}, N = 96$ corresponding to the 96 sensors observed in the simulations data, and by the set of edges $E = \{(v_1, v_2) | v_1, v_2 \in V \text{ and } dist(v_1, v_2) \leq d_{max}\}$ which links nodes with the maximum Euclidean distance d_{max}. This later has been set equal to $\sqrt{3}$, the diagonal of a cube with a 1-meter side, in order to model the neighborhood of the simulated sensors grid.

With the aim of consistently comparing the results obtained with the ones given by the other methodologies considered, the set of sensor nodes V is split into 2 subsets: $V_{test} = \{10, 11, 18, 33, 34, 51, 76, 78, 92\}$, a total of 10% reserved sensors randomly chosen for performance evaluation purposes, and $V_{train} = V \setminus V_{test}$, the set of sensors actually employed in the learning phase. The resulting graph $G_{train} = (V_{train}, E_{train})$ is obtained by restricting the set of edges E only to nodes in V_{train} (i.e., $E_{train} = \{(v_1, v_2) | (v_1, v_2) \in E \text{ and } v_1, v_2 \in V_{train}\}$). The aforementioned graph attention-based procedure is applied to G_{train} with the temperature values of $v_i \in V_{train}$ in Celsius (°C) as node feature $\mathbf{h}_i \in \mathbb{R}$, and the coordinates differences $\mathbf{h}_{i,j} \in \mathbb{R}^3$ between the nodes v_i, v_j on the 3 axes (z, y, x) as edge features.

The considered winter sequence Seq_w concerns 3183 samples (time instants), which means that for each sensor in the grid there are 3183 observed temperature values. In order to tune the hyperparameters for the GAT model, 10% of these ran-

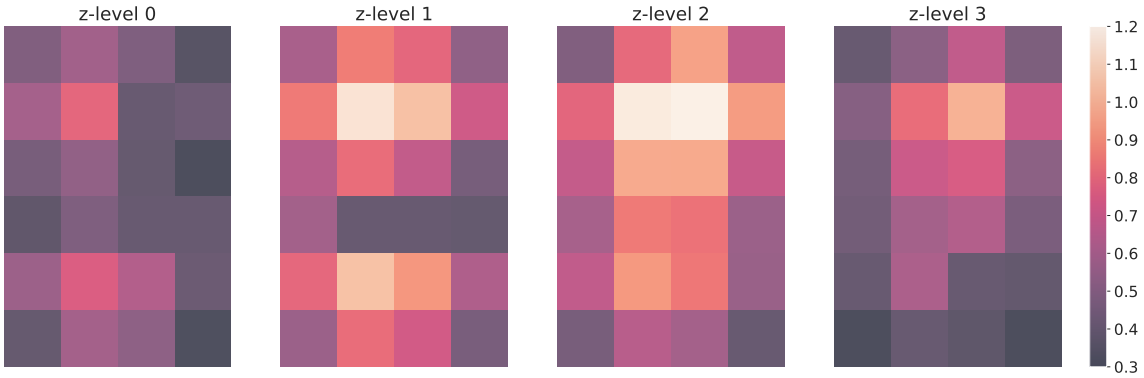


Figure 3.22: Graph attention-based score mapping for room configuration 1 on different z axis levels.

domly chosen samples are reserved for the validation set. In particular, the tuning of the hyperparameters took place via grid search optimization considering the following intervals: *learning_rate* in $\{0.1, 0.01, 0.001\}$, *weight_decay* in $\{0.1, 0.01, 0.001\}$, size of node embeddings F' in $\{1, 2, 4, 8, 16, 32\}$, size of edge embeddings F'_e in $\{1, 2, 4, 8, 16, 32\}$. As a result of the tuning phase, the hyperparameters have been set with the values 0.001 for *learning_rate*, 0.01 for *weight_decay*, and 8 for F'_v. The training of the GAT model is performed through the Adam optimization algorithm minimizing the *mean_squared_error* loss function based on the temperature values actually observed and those predicted for each sensor in V_{train}. Additional hyperparameters values adopted for the training phase are the *batch_size* set to 16, the momentum and RMSProp terms $\beta_1 = 0.9$ and $\beta_2 = 0.999$, and the training *epochs* equal to 200.

3.2.5 Results

Here we present the results obtained by the proposed graph attention-based ranking procedure, comparing them with those observed by applying a method outlined in the previous section, the weighted Borda count ranking (see Section 3.1.3), and a baseline solution based on a random ordering of the considered sensors.

In Figure 3.21, the results obtained by carrying out the experimentation described in the previous subsection are shown. Here the sensors in V_{train} (x axis) are sorted in descending order with respect to their achieved score (y axis). Applying the Kneedle algorithm, the n_{refs} value equal to 9 corresponding to the elbow of this curve is found. The final output of this procedure is the set of selected sensors $V^{att}_{refs} = \{66, 65, 41, 29, 42, 90, 61, 62, 70\}$. Furthermore, in Figure 3.22 the score values are mapped in 4 partition levels of z axis. This mapping shows that the proposed

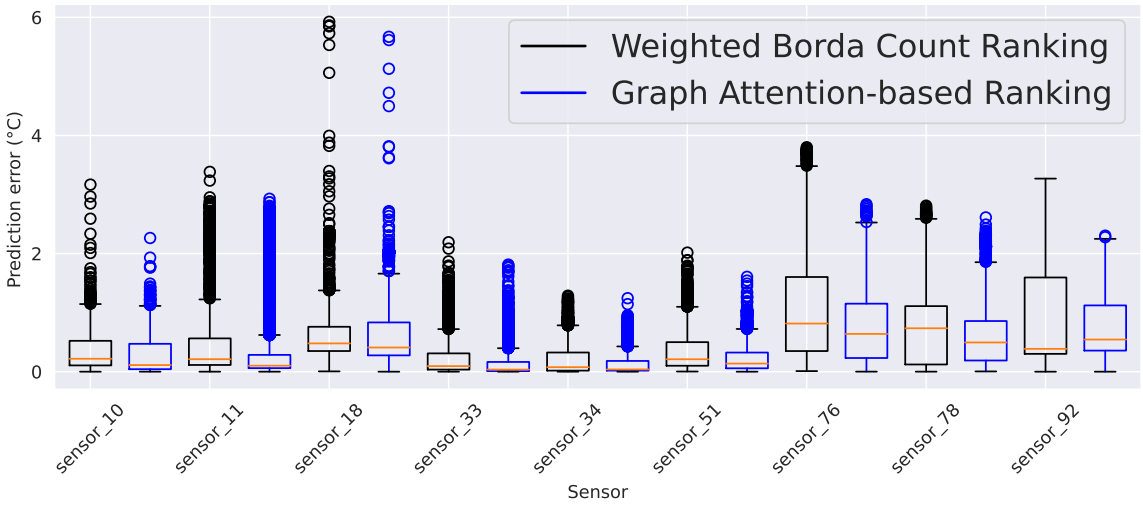


Figure 3.23: Boxplots of prediction errors provided by graph attention-based and weighted Borda count solutions.

procedure privileges points located in the central area with respect to the coordinates x, y, z, near the windows and the door. Although the procedure selected also nearby sensors such as 61, 62, and 65, 66, this provides several advantages. When sensors are positioned close to each other, local temperature fluctuations are likely to be similar among the sensors. This implies that sensor-specific errors, such as noise or drift, are likely to be similar as well. By utilizing multiple measurements from nearby sensors and combining them, it is possible to reduce uncertainty and obtain a more accurate estimation of the actual temperature. Furthermore, these sensors, albeit close in proximity, are spatially distributed accounting for the thermal characteristics of the surrounding area. Placing sensors strategically can provide better coverage and representation of temperature variations. In our case, sensors 61 and 62 are located near heater 1 and the partial wall, while sensors 65 and 66 are situated close to the air vents and window $w2$.

For the considered case, the set of sensors selected with the procedure based on weighted Borda count of Section 3.1.3 is $V_{refs}^{borda} = \{61, 38, 57, 65, 41, 58, 42, 66, 60\}$. Briefly, this method computes the rank scores by exploiting orderings based on a distance metric between nodes obtained through a genetic algorithm capable of synthesizing 3 spatial distance measures: Chebishev, Manhattan and Euclidean. For each sensor $v_i \in V_{train}$ a ranking of the remaining $|V_{train}| - 1$ sensors is computed by sorting these latter in ascending order according to their genetic distance from sensor v_i. The final scores are given exploiting the Borda count vote mechanism weighted considering the error given by each ranking for the prediction of the temperature in v_i.

Compared to the graph attention-based method, the sensors selected by means of the weighted Borda count procedure are placed towards the first half of the rank. As for the room coverage, they are located in the north-central area in front of the stoves $1, 2$ and H. This indicates that the attention-based graph ranking method tends to provide greater coverage of the room area, also considering elements which do not directly generate air flows.

The comparative tests were carried out by training two XGBoostRegressor[41] models to predict the temperatures of test sensors in V_{test}: the former makes use of the temperatures observed by the sensors in V_{refs}^{att} as input data, while the latter those from sensors in V_{refs}^{borda}. We chose to adopt these models for the comparison considering the limited usage of computational resources and their ease of training. The obtained results can be found in Figure 3.23 where boxplots include the prediction errors for each test tensor. The whiskers correspond respectively to the minimum and maximum errors excluding the outliers ($<$ 1st quartile - 1.5 IQR or $>$ 3rd quartile + 1.5 IQR, where IQR is the interquartile range given by the difference between the 3rd and the 1st quartile), while the orange line represents the median of the errors. As can be seen from the boxplots, the attention-based ranking graph shows better performance as regards outliers, while the two solutions are substantially equivalent as regards the distribution of prediction errors of lower magnitude.

A further evaluation of these sensor selections procedures was carried out considering as baseline a set of random combinations of 9 sensors in V_{train} extracted with different seeds. This is done in order to evaluate whether the sensors selected using weighted Borda count and graph attention perform better than a casual selection of equal cardinality ($|V_{refs}^{att}| = |V_{refs}^{borda}| = 9$). Since the total number of combinations is equal to $\binom{|V_{train}|}{9} = \frac{87!}{(87-9)!9!}$, evaluating all of them is intractable. Therefore, a sample of 1000 drawings generated with different seeds is considered. For each generated combination, an XGBoostRegressor model is trained using the extracted sensors as predictors for the temperatures of test sensors in V_{test}. As an outcome of the latter evaluation we consider the distribution of the prediction error ϵ averaged over these latter 1000 trained models.

The values reported in Table 3.1 are the outcome of the Wilcoxon signed-ranked test [183] comparing for each test sensor in V_{test} the distributions of the errors computed for the three methods in comparison. As can be seen, for this setting graph attention-based ranking performs better than weighted Borda count. Moreover, both of the latter techniques exhibited better results when compared to a pure random combinations-based solution. In order to control the family-wise error rate of the multiple hypothesis tests, the last row of Table 3.1 reports the Holm-Bonferroni [80] adjusted p-values.

Table 3.1: Sensor selection methods error distributions compared through Wilcoxon signed-rank test.

Test sensor		**Random Combinations Vs. Weighted Borda**	**Weighted Borda Vs. Graph Attention**
Sensor 10	stat.	$1.762e+006$	$1.288e+006$
	p-value	$2.480e-050$	$7.112e-128$
Sensor 11	stat.	$1.875e+006$	$1.158e+006$
	p-value	$2.820e-037$	$2.506e-155$
Sensor 18	stat.	$1.496e+006$	$1.824e+006$
	p-value	$1.821e-089$	$5.981e-043$
Sensor 33	stat.	$2.031e+006$	$8.164e+005$
	p-value	$1.587e-022$	$7.806e-241$
Sensor 34	stat.	$1.476+006$	$1.378e+006$
	p-value	$8.111-093$	$2.657-110$
Sensor 51	stat.	$1.273e+006$	$1.387e+006$
	p-value	$6.417e-131$	$1.366e-108$
Sensor 76	stat.	$2.380e+006$	$8.722e+005$
	p-value	$1.552e-003$	$1.230e-225$
Sensor 78	stat.	$2.361e+006$	$1.866e+006$
	p-value	$4.339e-004$	$3.274e-038$
Sensor 92	stat.	$1.850e+006$	$1.802e+006$
	p-value	$4.891e-040$	$1.521e-045$
Holm-Bonferroni Correction	p-value	0.006	0.006

Note: the performed test is one-sided with alternative hypothesis defined such as the distribution underlying $d = X_i - Y_j$ is stochastically less than a distribution symmetric about zero, where X_i is the absolute prediction errors given by the first compared method, while Y_i the absolute prediction errors given by the second one.

3.2.6 Discussion

Besides providing better performance in terms of predictive model accuracy, the sensor selection solution proposed in this section has further advantages when compared to the one based on weighted Borda count. First of all, it is more efficient in terms of the number of trained models while performing the procedure. In particular, considering a set V of candidate reference sensors, the method based on graph attention requires the training of $|V|$ trained sensors using only the information provided by a small subset of sensors positioned near of each of those to be predicted. Instead, the weighted Borda count procedure requires the creation of $|V| * (|V| - 1)$ models trained considering the information coming from all the sensors present in the subset of reference candidates considered. Secondly, a graph attention network-based solution has the ability to learn complex patterns and relationships among sensors in the environment. Furthermore, a graph attention network-based solution

can scale to larger sensor grids with more sensors and complex topologies, whereas a genetic programming generated spatial distance metric-based solution may struggle to handle the increased complexity. Finally, it is worth noticing that, being based on genetic programming, the weighted Borda count solution may be more interpretable, as the generated distance metrics can be directly analyzed and understood. However, a graph attention network-based solution can also provide some level of interpretability through visualization of the attention weights.

To conclude with a brief summary, the sensor placement procedure based on graph attention networks described in this section is applied starting from simulation data generated through a sufficiently dense sensing monitor grid. This latter returns a list of reference positions whose observed temperature values allow the training of a predictive model acting as an efficient simulation approximator. This is done by encoding the environmental information in a graph structure fed as input during training as we shall see in Section 4.2.1.Furthermore, in Section 4.2.2 we propose and analyze an algorithm capable of transferring the selected set of sensors to a different room configuration.

4

Prediction

In the previous chapter of this thesis, we addressed the problem of sensor selection for temperature monitoring in indoor environments. In light of what has been achieved, in this chapter, we delve into the prediction of temperatures through virtual sensing techniques. Specifically, we compare the performance of virtual sensing strategies including novel solutions proposed for two different case studies: the real-world indoor environment, which was presented in Section 3.1.1, and the simulated indoor environment, presented in Section 3.2.1.

In the first section of this chapter, after an introduction to the problem domain, we focus on the virtual sensing of temperatures in the real-world indoor environment case, exploring various baseline methods, particle filters, machine learning approaches, and a deep learning approach. Here, we also conduct an analysis of prediction intervals and errors per single sensor. Additionally, we investigate the effects of reducing the training data size on the accuracy of the predictions.

Subsequently, in Section 4.2.1, we shift our attention to the monitoring of temperature problem in simulated indoor environments. Here, we compare the performance of some of the previously analysed inductive baseline methods and machine learning approaches with a novel proposed transductive learning solution based on graph neural networks. Our experimental results indicate that the latter is able to provide better performance by exploiting environmental information more effectively. We conclude the section analysing the behavior of the proposed predictive model, and the adaptability of the sensor placement procedure presented in Section 3.2.2 in case of different environmental conditions.

4.1 Virtual sensing of temperatures in real-world

As argued in Chapter 1, virtual sensing is a set of techniques to replace a subset of physical sensors by virtual ones, allowing the monitoring of unreachable locations, reducing the sensors deployment costs, and providing a fallback solution for sensors failures.

Over the years, virtual sensing has found application in several domains including robotics, automation, anomaly/leak detection, air quality control, active noise suppression, wireless communication, automotive, and transportation [109, 107, 163]. In the literature, three main virtual sensing modeling methods have been taken into account: (*i*) *first-principle*, where virtual sensors are developed by hand on the basis of fundamental laws of physics and an extensive domain knowledge; (*ii*) *black-box*, where models capable of exploiting empirical correlations present in the data are built, without any knowledge of the underlying physical processes (most of the statistical and machine learning methods belong to this category); (*iii*) *grey-box*, that exploits a combination of first-principle and black-box approaches. In this study, we focus on the latter two approaches, in order to develop generic models that do not require complex domain knowledge and are not tied to the specific use case.

As for black-box techniques, the estimation of a variable at locations where it has not been observed by exploiting data available at other locations can be made by means of approaches commonly used for the interpolation of scatter points, such as Inverse Distance Weighting (IDW) and Kriging [138]. IDW assumes that the interpolating surface is more influenced by closer points than distant ones, while Kriging is a geostatistical regression method used in spatial analysis that spatially interpolates a quantity minimizing the mean squared error. As it comes out, both approaches find difficult application in real-world settings, being them typically affected by spatial-temporal anisotropy and unable to deal with non-linear relations among predictor variables. When partial observations are made and random perturbations are present in the data, methods such as Kalman filters and particle filters [55] can be relied upon for the estimation of the internal states of dynamical systems. These methods compute the unknown quantities through posterior distributions obtained from Bayesian inference models [180, 127]. Although particle filters are not subject to the constraints of non-Gaussianity of perturbations and linearity of the dynamic systems that affect Kalman ones, they have some disadvantages as well, related to possible resampling biases and to the coarseness of the definition of the likelihood distribution, which may be unable to capture all relevant real-world characteristics. To overcome the limitations of these techniques, some authors explored approaches based on machine learning, like support vector machines, decision trees, and ensembles (such as random forests) [179, 102, 101]. More recently, deep learning solutions have been considered as well [156, 172, 186, 187], and were deemed able to exploit implicit information on temporal trends and spatial associations among sensors. As an example, in [113, 114] the authors employ respectively a Long Short-Term Memory (LSTM) network and a combined Convolutional LSTM (ConvLSTM) network capable of learning from long-term dependencies in spatial-temporal information. Despite the latter ConvLSTM-based model achieved a good performance,

it is not applicable in context like ours where the positions of sensors are fixed and quite sparse.

4.1.1 Temperature prediction

In this part of the section, to determine the performance of the sensor and feature selection phases presented in the Sections 3.1.3 and 3.1.4 with respect to the task of temperature virtual sensing, and to identify the best prediction methodology, we experiment and contrast the following approaches:

- Baseline: simple and Inverse Distance Weighted (IDW) average of the temperatures;
- Particle filters;
- LinearRegression – Python's package Scikit-learn [143];
- XGBoostRegressor – Python's package xgboost [41];
- An LSTM recurrent neural network, trained by means of the PyTorch Deep Learning library [142].

As pointed out in Section 3.1.3, except for the baseline methods and particle filters, predictions make use of the temperature values recorded by the 4 chosen reference sensors (sensors 1, 4, 8, and 9) depicted in blue in Figure 3.1. To ensure the comparability of the results obtained from the various approaches, prediction errors were evaluated on the remaining 8 sensors (the original 12, except the 4 reference sensors, that are already used as predictors). In addition, as previously mentioned, we always considered the same 80%-20% split training-test of the dataset.

The predictive analysis tasks are organized as follows in the remainder of this section. Firstly, we evaluate the various approaches to temperature virtual sensing. Then, we analyze the prediction errors assessing the uncertainty associated with the predicted quantities, investigating the errors per single sensor, and linking the prediction error to the available portion of training data. Finally, the outlined framework is evaluated with respect to the optimal result that can be achieved by means of a brute force approach to sensor selection.

Baseline methods

This first analysis allowed us to define a baseline against which to compare the results of the other approaches. Given a sensor for which we want to predict the temperature readings on the test set, the idea is that of approximating such values

by a simple combination of the temperatures recorded by the other ones at the same time instant. To this end, we applied two different techniques: classical average and Inverse Distance Weighted Average (IDWA), according to which closer sensors have an impact on the overall prediction greater than that of those sensors which are farther away. In more detail, the weight assigned to the i-th predictor is computed as $w_i = \frac{1}{d(x,x_i)}$, where $d(\cdot,\cdot)$ is the genetic programming-based distance (Section 3.1.2) between two points of the grid, x_i is the position of the i-th sensor, and x is the position of the sensor to predict. The weighted temperatures are summed and then divided by the sum of the weights.

Moreover, to determine the impact on the prediction accuracy of the distance between sensors, we performed several experiments by considering as predictors just the k sensors closest to the one to predict, for $k \in [1, 11]$. For each approach and sensor to evaluate, we determined the temperature absolute error, considering its 95th percentile, ϵ_{95}, which can be thought of as a worst-case prediction scenario. Figure 4.2 collects the boxplots of ϵ_{95}. Each boxplot includes a value for every test tensor, for a total of 8 values. The orange line represents the median of ϵ_{95}, while the whiskers correspond respectively to the minimum and maximum values excluding the outliers ($<$ 1st quartile - 1.5 IQR or $>$ 3rd quartile + 1.5 IQR, where IQR is the interquartile range given by the difference between the 3rd and the 1st quartile). It clearly emerges that classical average is largely influenced by the number of closest sensors used for the prediction. Here, the optimal number of sensors seems to lie in the range $[3, 5]$. On the other hand, IDWA seems to be less affected by the number of predictor sensors. Indeed, looking at the median, large values of k led to better results. This is to be expected, as the contributions of the different sensors are already weighted according to their distances.

Particle filters

Particle filters are a class of Sequential Monte Carlo algorithms used to approximate the internal states of dynamic systems starting from partial measurements with random disturbances which afflict the sensors as well as the dynamic system itself [55]. Given the noisy and partial observations, this approach aims at measuring the state posterior distributions of some Markov process. Particle filters leverage a set of particles to represent such a posterior distribution. Each particle has an assigned weight, indicating the chance of that particle being sampled from the probability density function of the quantity we want to compute. As for this experimentation, we considered one particle filter for each of the 8 evaluated sensors, and the k-closest sensors as landmarks, with $k \in [1, 11]$. Each particle thus represents a likelihood estimation of a temperature and is moved at each time-step following the average

Table 4.1: XGBoostRegressor parameters (rounded to the 5th decimal digit).

Parameter name	*max_depth*	*learning_rate*	*n_estimators*	*reg_alpha*	*reg_lambda*	*gamma*	*subsample*	*colsample_bytree*	*min_child_weight*
Value	16	0.015	350	78.87396	0.50044	5.95353	0.66425	0.65694	1.0

temperatures of the landmarks. The likelihood probability is computed on the basis of the genetic programming distance between particles and landmarks.

As with the baseline approaches, we estimated the error's 95th percentile for each evaluation sensor and each potential k value. Results are reported in Figure 4.2. It is evident that also particle filters are affected by the choice of k. For the sake of readability, we decided to ignore the results for $k \in [1, 3]$, as those values led to very weak predictions, with boxplot whiskers extending over 6 degrees. As for the remaining values, according to the median, the best predictions are achieved for values of k equal to 4 or 5.

Machine learning approaches

We considered two different machine learning approaches, namely, a simple Scikit-learn's LinearRegression model, and a more complex XGBoostRegressor ensemble approach.

LinearRegression implements an ordinary least squares linear regression. For the sake of our study, it has been trained on the 11 (standardized) features selected in Section 3.1.2, with training labels corresponding to the temperatures recorded by 8 evaluation sensors (the original 12, except the 4 reference sensors, that are used as predictors).

XGBoost [41] implements gradient boosted decision trees focusing on computational speed and model performance. Gradient boosting iteratively builds new models to predict the residuals of errors of previous models exploiting a gradient descent algorithm to minimize the loss [63]. The resulting models are then combined to generate the final prediction. As a first step, we tuned the XGBoostRegressor model with the above-described training set, from which a validation set of size 20%, consisting of randomly chosen weeks, was extracted. The task was performed by means of *Hyperopt* [15], a library for hyperparameter optimization written in Python, minimizing the 95th percentile of the error loss function for 40 evaluation steps on the following hyperparameters: *max_depth*, *learning_rate*, *n_estimators*, *reg_alpha*, *reg_lambda*, *gamma*, *subsample*, *colsample_bytree*, and *min_child_weight*. Resulting values are listed in Table 4.1. With the tuned hyperparameters, the model was trained on the entire training set, and then evaluated on the test set over the usual 8 sensors.

The outcomes shown in Figure 4.2 suggest that the tested machine learning

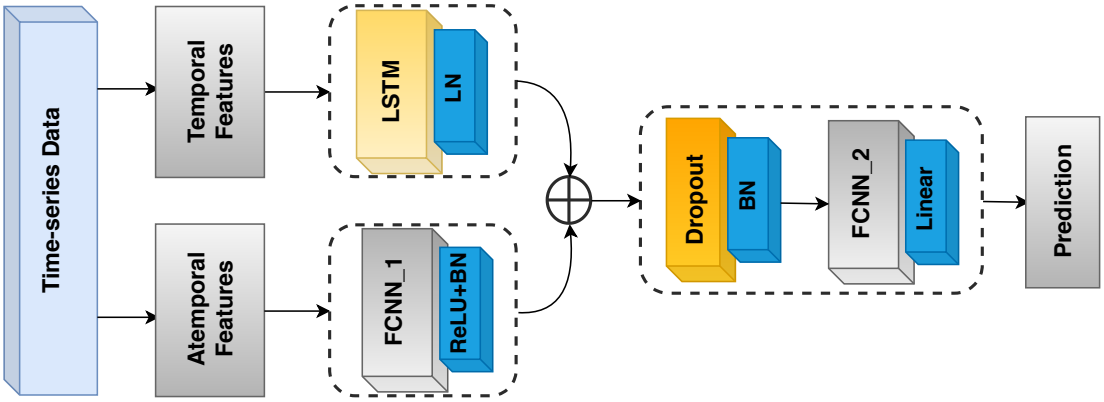


Figure 4.1: The LSTM recurrent neural network model architecture.

methods vastly outperform the baseline approaches. Specifically, XGBoost shows a better performance than LinearRegression on all 95th error quantiles. Furthermore, XGBoost's boxplot is wider than that of LinearRegression, suggesting a more unstable behavior across the sensors predictions. This is to be expected, being XGBoost a far more complex and flexible model.

Deep learning approach

Up to this point, to predict the temperature at a given time instant we have just considered reference sensor values from the same instant. In the literature on remote and virtual sensing, it has already been shown that deep learning methods are able of taking temporal and spatio-temporal knowledge into account (see, e.g., [156, 172, 186]). Specifically, in our context, it may be the case that the recent history of temperatures reported by the reference sensors provides information relevant to the overall prediction. As an example, opening a window in winter time, one may notice a regular and continuous decrease of the temperatures recorded by a nearby temperature sensor. This, together with information recorded by the other reference sensors, may give the model a hint about the temperature propagation in the room. We designed an LSTM-based model that takes such histories into account. Its architecture, which is depicted in Figure 4.1, consists of three subparts:

- the first (*temporal*) part (LSTM on the upper left side of Figure 4.1) takes a history of the 4 (standardized) reference temperatures as input. Then, a unidirectional LSTM layer, consisting of 128 units, from which we retrieve just the last outputs, followed by LayerNormalization, is applied;
- the second (*atemporal*) part (FCNN_1 on the bottom left side of Figure 4.1) takes the 7 remaining (standardized) attributes as input, resulting from the

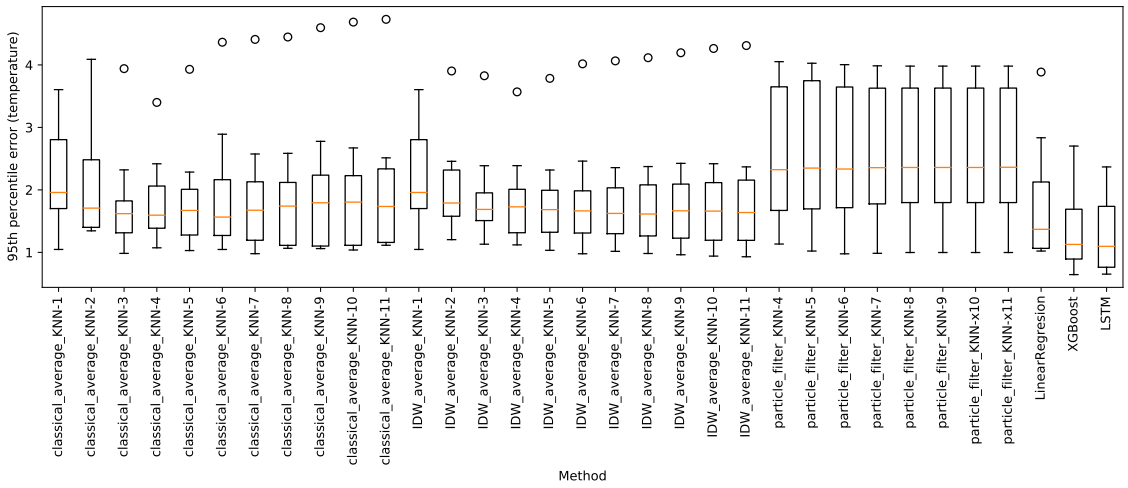


Figure 4.2: Boxplots of the 95th percentile of the error provided by the considered approaches.

feature selection process (Section 3.1.2). These attributes do not have any significant history, but are still important to generate the final output, since they allow the model to pinpoint the prediction in space and time. The aforementioned 7 features are passed to a Dense layer, consisting of 64 neurons, followed by a ReLU activation function and a BatchNormalization layer;

- the third part (FCNN_2 on the right side of Figure 4.1) takes the outputs of the first two parts and concatenates them, generating a tensor of size 192. Then, BatchNormalization and Dropout with 0.1 rate are applied to the result of such a concatenation. Next, the data goes through a dense layer of 128 units, followed by a ReLU activation function, and a BatchNormalization layer. The final output is produced by a single-unit Dense layer with linear activation function.

In order to train the neural network, we relied on Adam optimizer with a 9e-5 learning rate minimizing the mean_squared_error loss function. The architecture of the model and all the other hyperparameters have been chosen through iterative random search tuning performed on a fixed 80%-20% training-validation subsplit, as already done for the previous machine learning approaches. At last, the same tuning process suggested a length of 18 samples (equivalent to a period of 3 minutes) for the reference temperature histories, over a tested range of 1 to 5 minutes.

As shown in Figure 4.2, with respect to the test set, the network essentially provided the same performance as XGBoostRegressor: while the upper whisker is marginally better than that of the ensemble approach, the lower one is slightly

worse. Furthermore, the broader extension of the boxplot suggests a more variable behavior than XGBoost. Based on these results, perhaps surprisingly, we can infer that, in our setting, historical knowledge of temperatures alone does not contribute much to the accuracy of the prediction. This can be justified by the fact that the sensors we want to predict are typically located much closer to a window than the reference ones, and thus they should also be the first to be affected by any weather-related phenomenon. Accordingly, reference sensors' historical data are not so important from this point of view. As an additional confirmation, results of an auto-correlation analysis showed that information conveyed by the reference sensors at the time instant in which the prediction is carried out is way more relevant than information present in the historical temperature values related to the same sensors.

In future work, we plan to evaluate the performance of a CNN-LSTM-based neural network that, in principle, should allow us to relate the temporal dimension of temperature histories to spatial information about the placement of the reference sensors, the distances among them, and their distance from the location we want to predict.

Prediction intervals analysis

Usually, a regression model generates a single value for each prediction, which represents itself a random variable. However, under several circumstances, quantifying the uncertainty associated with the prediction, instead of computing just a single value, is very useful, as it gives an indication of the reliability of the results. This can be done by setting proper prediction intervals. Such intervals provide probabilistic upper and lower bounds on the estimate of an outcome variable and can be computed via quantile regression [96]. Typically, regressions minimize the mean squared-error loss function $L(y, \hat{y}) = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y_i})^2$, while quantile regression aims at estimating conditional quantiles of the response variable. This is achieved by adopting the loss function $L_\gamma(y, \hat{y}) = \sum_{i=y_i<\hat{y_i}}^{N}(\gamma-1)|y_i-\hat{y_i}|+\sum_{i=y_i\geq\hat{y_i}}^{N}(\gamma)|y_i-\hat{y_i}|$, where N is the number of the samples in the training set, γ is the quantile of the response variable to forecast, $\hat{y_i}$ is the predicted value for the i-th sample, and y_i is the real target value for the i-th sample.

We explored two different approaches to quantile regression: a linear regression model and a gradient boosting regression model [63]. In both cases, two models were trained: one for the upper ($\gamma = 0.025$) and one for the lower ($\gamma = 0.975$) bound of the interval. This means that 95% of the actual values should lie between these two predicted bounds. The training procedure was the same as in Section 4.1.1 and an excerpt of the results for the sensor 3 test data is shown in Figure 4.3, in the case of linear regression, and in Figure 4.4, in the case of gradient boosting

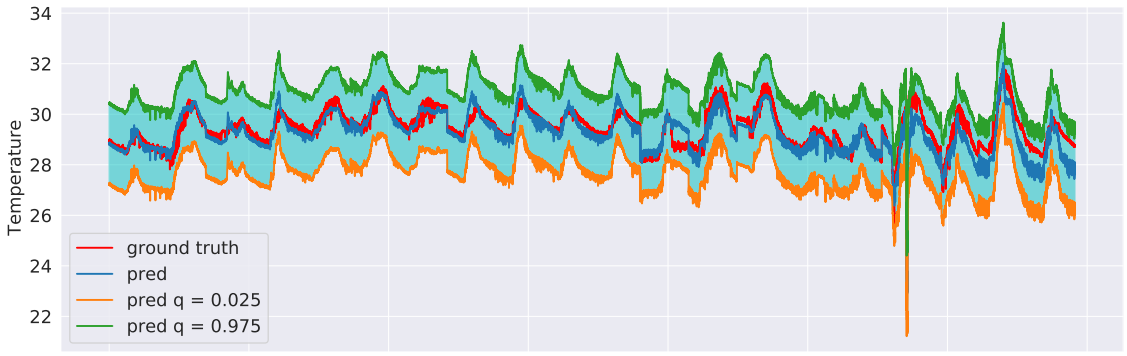


Figure 4.3: Linear regression prediction intervals related to sensor 3 test data with $\gamma = 0.025$ and $\gamma = 0.975$.

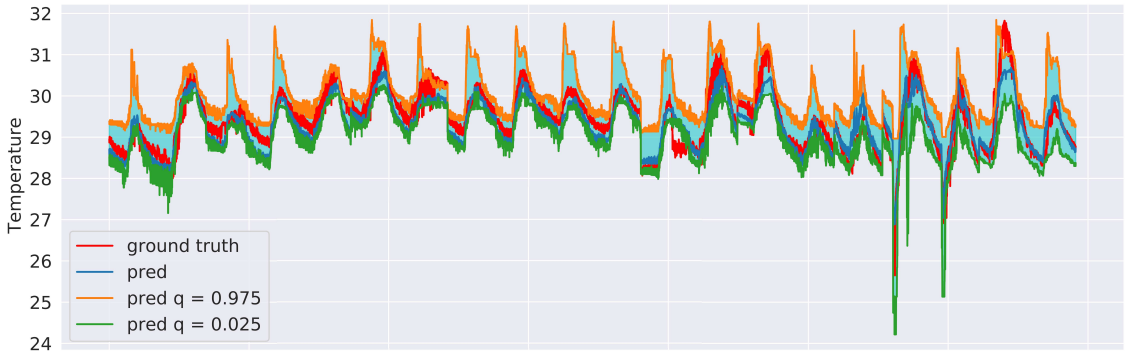


Figure 4.4: Gradient boosting regression prediction intervals related to sensor 3 test data with $\gamma = 0.025$ and $\gamma = 0.975$.

regression. Although both approaches were indeed observed to guarantee that 95% of the observed values end up between their estimated lower and upper bounds, the intervals obtained from the Gradient Boosting models are generally less coarse, thus providing a better approximation of the uncertainty intervals.

Errors per single sensor

Let us now take a closer look at the performance of the machine and deep learning approaches. Figure 4.5 reports the 95th percentile errors of such models for each of the 8 evaluation sensors. Although the LSTM and XGBoost models typically exhibit a better performance than LinearRegression, the latter, despite being a much simpler model, is a close match, with the notable exception of sensor 10, where it is vastly outperformed by its contenders. It is also worth observing the relatively high error rates on sensor 7: upon closer inspection, this sensor displayed a very strange,

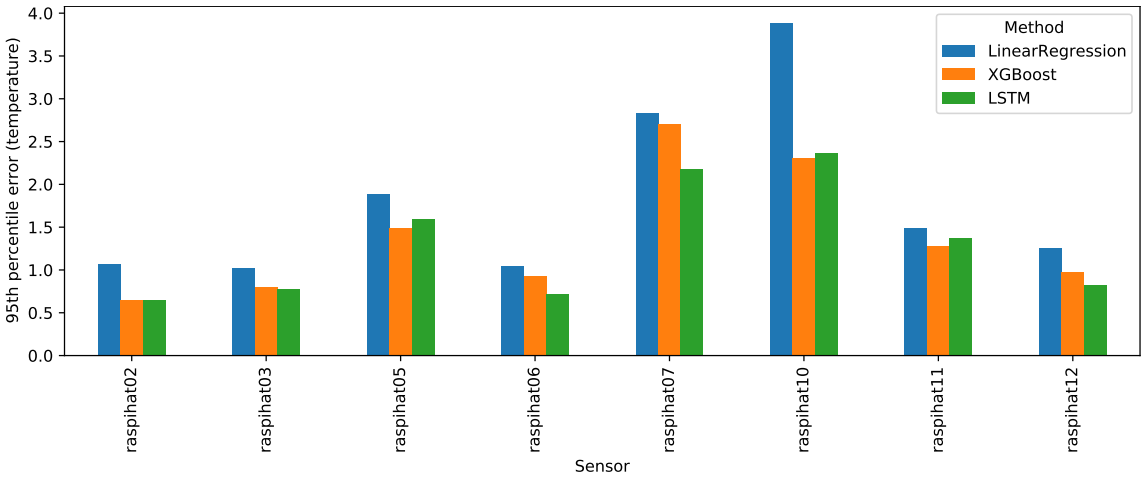


Figure 4.5: Results of machine and deep learning approaches for each evaluation sensor.

fluctuating curve, as shown in Figure 4.7. It may be difficult to predict this kind of irregular temperature values using a global model for all 8 sensors, as done in this section. Therefore, as a future analysis, we plan to compare the outcomes of the global models with those that can be obtained by building a single model for each sensor to be predicted.

Effects of reducing the training data size

Given the large quantity of training data available, we examined the effects of reducing its size, relying on the XGBoost model introduced in Section 4.1.1. We tested various training set cardinalities obtained by sampling the data at one week granularity. The experiment was conducted using 10 sampling rates of $r \in \{0.1, 0.2, \ldots, 1.0\}$, repeating each execution with 10 different random seeds to prevent sampling bias. As expected, the findings in Figure 4.6 show that, as the training data size decreases, the median of the 95th percentile of the errors raises, probably due to the fact that some seasonal trends may be overlooked if a too small data sample is employed.

To further investigate the prediction error related to the outliers, we iteratively discarded the test data belonging to different devices, and it ultimately emerged that the anomalous values belonged to the predictions made for sensor *raspihat07*. Figure 4.7 shows the temperature values of sensor *raspihat07* and of its 3 closest neighbors (*raspihat01, raspihat04*, and *raspihat08*) limited to the time instants at which outliers are present in the prediction error (*mean absolute error* $\geq 2°$C). At these time instants, sensor *raspihat07* shows a more marked fluctuation behavior for both high and low values, that may suggest a degradation or bad calibration of

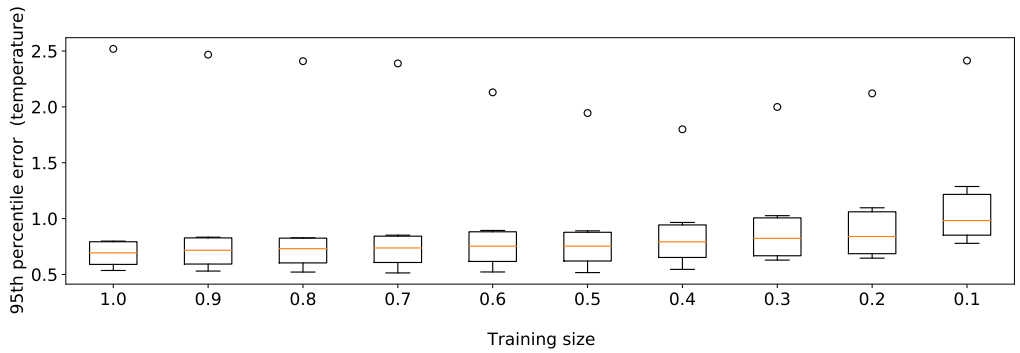


Figure 4.6: XGBoost results obtained by varying the training set size.

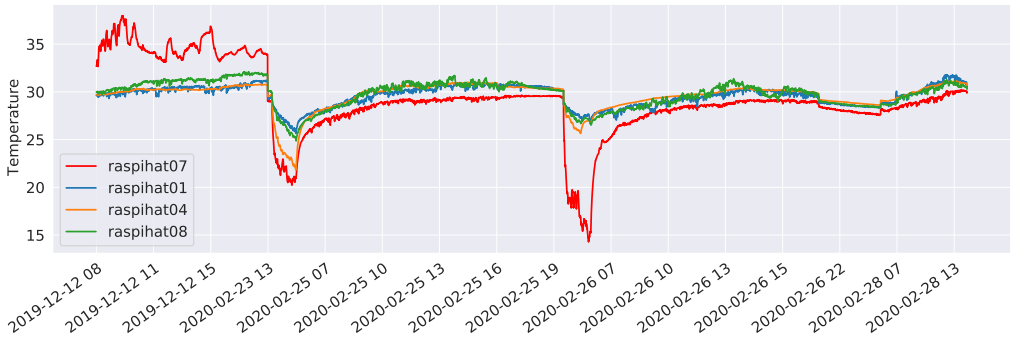


Figure 4.7: Sensor *raspihat07* temperature values related to the prediction error outliers compared with the 3-nearest neighbours sensors.

the device. Finally, we reran the training data size reduction experiment discarding the data related to sensor 7 from the evaluation. In this way, a more predictable monotonic decrease of the prediction performance was observed.

4.1.2 Discussion

In the previous parts of this section, we focused on how to perform virtual sensing efficiently, dealing with the problem under different points of view. Our proposed solution encompasses all the relevant aspects of virtual sensing, including sensor selection, the estimate of the needed amount of training data, the choice of the predictive model, and the evaluation of its performance. Most importantly, the approach can be regarded as a *black-box*, completely independent of the physical characteristics of the considered scenario, such as any element capable of influencing the internal temperature (windows, radiators, and so on). Thus, in principle it can be applied to any generic indoor environment with an arbitrary set of pre-located sensors.

4.2 Virtual sensing in simulated indoor environments

In this section, we focus on the task of temperature monitoring in a general indoor environment, where temperature data is provided through a set of physical simulations. Our main goal is to develop a temperature prediction framework through virtual sensing and spatial interpolation techniques able to model the physical characteristics of the considered scenario, and to adapt to other, similar environments.

Hereafter, several solutions available in the literature are examined highlighting their strengths and limitations when applied to the considered scenario.

A family of methods able to deal with the indoor thermal monitoring problem is the one given by classical spatial interpolation approaches such as inverse distance weighting, nearest neighbor, spline and Kriging. Applications of these methods can be found in the literature for the case of data center thermal monitoring [138], indoor environmental thermal and quality distribution [190, 42, 188], or radio environment [53] mapping. Although easy to implement, these approaches, can only be used as a baseline, as they are based on limiting assumptions, such as non anisotropy or simple linear (and non-linear) relations among predictor variables, and are typically inaccurate when applied within complex settings. To overcome some of these limitations, [100] presents an approach which separates spatial and temporal information combining local autoregressive prediction based on past observations, with spatial interpolation. Since local forecasts can be parallelized, this latter has the advantage of being efficiently computed. Furthermore, the subsequent application of spatial interpolation via Kriging ensures a good accuracy. Conversely, this approach relies on assumptions, such as isotropy and independence of spatial and temporal phenomena, that may not hold in some scenarios.

As for virtual sensing, we consider the three modeling methods introduced in Section 3.1, namely, (*i*) *first-principle*, (*ii*) *black-box*, and (*iii*) *grey-box*.

In the case of *first-principle* methods, techniques such as computational fluid dynamics [6] are employed to generate simulations starting from a specification of the considered indoor environment and the elements placed into it. This type of approach, exploited in this work to produce the ground truth simulation data, is based on mathematical modeling through laws of physics describing physical phenomena. For this reason, they are particularly accurate in measuring characteristics of these simulated phenomena. On the other hand, since they typically rely on the solution of differential equations, they are computationally expensive, especially for real time applications concerning complex environments. Even if some lightweight solutions, based on order reduction of the energy balance equation through proper orthogonal

decomposition (POD), have been proposed [170, 58], these models are tailored to a given specific environment and difficult to implement. Furthermore, generating physical simulations with this kind of approaches is an expensive activity in terms of effort required for the creation of numerical models for different environmental configurations.

Rather, we are interested in approaches able to learn how to model physics dynamics and characteristics of the environment to be monitored, allowing the resulting predictive models to generalize to other environments with different configurations and conditions.

A step in this direction is represented by physics-informed neural networks (PINN) [152], a *grey-box* approach in which the training process is guided imposing an inductive bias that adheres to the physics of the system at hand, described by means of partial differential equations (PDEs) to solve. As an example, for the case of heat diffusion, this is done by adding a regularization term like $\mathcal{L}_{\text{PDE}} = \frac{1}{N}\sum_{i=1}^{N}\left\|\left[\frac{\partial}{\partial t} - D\left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}\right)\right]\hat{u}(t^{(i)}, x^{(i)}, y^{(i)})\right\|^2$ to the loss function, penalizing for instance the solutions that do not satisfy the PDE $\left[\frac{\partial}{\partial t} - D\left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}\right)\right]u(t, x, y) = 0$, where N is the number of samples, D is the thermal diffusivity constant, $u(t, x, y)$ and $\hat{u}(t, x, y)$ are the functions returning, respectively, the real and predicted temperatures in position (x, y) at time step t. An application of this solution to the thermal modeling of buildings and heat transfer problems is presented in [70] and [30]. Despite being able to simulate physical processes with good accuracy (if compared with common fully connected neural networks), this family of models still requires a manual design phase and hardly generalizes to complex environments or configurations that are different from those examined during training.

For the sake of completeness, we point out that other solutions based on machine learning, like support vector machines, decision trees, and ensemble methods (such as extreme gradient boosting and random forests) have been explored by different authors [102, 101]. As for classical deep learning solutions, in [156, 172, 113, 114] a multi-layer perceptron (MLP) neural network, a Long Short-Term Memory (LSTM) network and a Convolutional LSTM (ConvLSTM) network able to exploit inferred information on spatial and temporal associations among sensors are presented.

Tipically, the aforementioned deep learning approaches require a retraining of the models or a fine-tuning phase in case of changes in environmental conditions or configurations. In contrast to all the above mentioned solutions, we propose an approach based on graph networks as approximators of physical simulations via learned message-passing functions where room elements, such as furnishing items or air flows sources, are represented by nodes in a graph. Such an approach has already been applied in the literature to challenging domains, involving fluids, rigid

solids, and deformable materials [164] proving its ability to learn how to simulate complex physics. Despite being focused on the temperature monitoring scenario, the considered approach can, in principle, be applied to any other sensor prediction task linked to indoor physical dynamics.

4.2.1 Temperature prediction

In this part of the section, our main focus is the evaluation of the accuracy in approximating the physical simulations with respect to the task of temperature virtual sensing and to identify the best prediction methodology. With these objectives in mind, we begin the analysis by describing and evaluating some state-of-the-art inductive learning approaches, including simple and inverse distance weighted (IDW) average of the temperatures as baseline, and XGBoostRegressor (using Python's package xgboost [41]) as machine learning approach.

Thereafter, considering the nature of the context which pertains to a room with a grid of sensors and various elements such as walls, heaters, windows, and doors, we aim to study the benefits of models able to represent spatial relationships between sensors and other room elements, and exploit this information to make accurate predictions of the temperature in each part of the room. With this premise, we propose a transductive learning graph network model (based on the PyTorch Geometric graph neural networks library [60]). Representing the room as a graph where nodes correspond to internal elements, and edges represent interactions between them. Together with a comparison of the graph networks-based model with the aforementioned inductive approaches, a systematic assessment of the impact, in terms of improved accuracy, of incorporating information from various sets of elements characterizing the indoor environment is carried out.

As pointed out in Section 3.2.2, predictions make use of positions and temperature values captured by 9 chosen reference sensors in V_{refs}^{att} as depicted in Figure 3.21. Prediction errors were evaluated on a specific set V_{test} of the remaining sensors, defined as $V \setminus V_{refs}^{att}$. Furthermore, to ensure the comparability of the results obtained from the various approaches, the set of dataset timesteps $T = \{0, \ldots, 3183\}$ has been divided in T_{train} and T_{test} according to an 80%-20% training-test split. Therefore, the accuracy of the considered approaches has been evaluated on the temperature values observed at time instants in T_{test} for sensors in V_{test}.

In the remainder of this section, we evaluate the various approaches of temperature prediction, and analyze the obtained results.

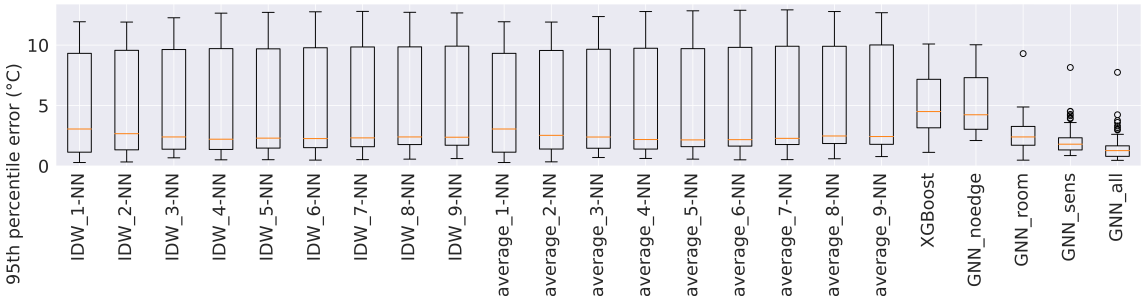


Figure 4.8: Boxplots of the 95th percentile of the error provided by the considered approaches.

Baseline methods

This first analysis allowed us to define a baseline against which to compare the results of the other approaches. Given a sensor in V_{test} for which we want to predict the temperature readings on the test set, the idea is that of approximating such values by a simple combination of the temperatures recorded by the other ones in V_{refs}^{att} at the same time instant. To this end, as conducted in Section 4.1.1, we applied two different techniques, namely, classical average and inverse distance weighted average (IDW).

Operationally, to determine the impact on the prediction accuracy of the distance between sensors, we performed several experiments by considering as predictors just the k sensors closest to the one to predict, for $k \in [1, |V_{refs}^{att}|]$. For each approach and sensor to evaluate, we determined the temperature absolute error, considering its $95th$ percentile, ϵ_{95}, which can be thought of as a worst-case prediction scenario. In fact, by focusing on the upper tail of the error distribution, this metric provides a more conservative estimate of model performance, which is particularly relevant in applications where large prediction errors can have significant consequences.

Figure 4.8 collects the boxplots of ϵ_{95}. Each boxplot includes a value for every test tensor in V_{test}, for a total of 87 values. Considering median values of ϵ_{95}, it clearly emerges that both classical average and IDW solutions are influenced by the number of closest sensors used for the prediction. The optimal number of sensors lies in the range $[3, 5]$. Overall, both baseline methods behave similarly and exhibit rather high upper quartile values, which indicates that they are unable to grasp complex relationship between the data. Furthermore, they follow a bowl pattern, a sign that the use of an excessive number of predictors leads to noise in the prediction phase.

Table 4.2: XGBoostRegressor parameters (rounded to the 5th decimal digit).

Parameter name	*max_depth*	*learning_rate*	*n_estimators*	*reg_alpha*	*reg_lambda*	*gamma*	*subsample*	*colsample_bytree*	*min_child_weight*
Value	16	0.015	350	78.87396	0.50044	5.95353	0.66425	0.65694	1.0

Machine learning approach

In this experimentation, as described in Section 4.1.1, XGBoost has been considered as machine learning approach.

Operationally, an XGBoostRegressor model has been trained considering as input information the following room elements: the 9 reference sensors in V_{refs}^{att} as defined in Section 3.2.2, and the 17 room elements (including partial and perimeter walls) as depicted in Figure 3.16 for room configuration 1. Overall, for each of these elements we consider as input feature the observed (in case of sensors) or emitted (in case of heater, radiator or vent elements) air temperatures in Celsius (°C), the emitted mass flow in kilogram per second (Kg/s) and the element orientation angle with respect to the z axis expressed in radians, 12 Boolean features representing the one-hot encoding of the element type which could be one choice from all possible room element types defined as $\mathcal{T} :=$ *sensor* | *heater_non_convective* | *heater_convective* | *vent* | *door_closed* | *door_opened* | *window_closed* | *window_opened* | *wall_external* | *wall_internal* | *wall_partial* | *table*. Each feature in this resulting training set has been standardized among all timesteps in T_{train} before training.

As a first step, we tuned the XGBoostRegressor model with the above-described training set, from which a validation set of size 20%, consisting of randomly chosen timesteps, was extracted. The task was performed by means of *Hyperopt* [15], a library for hyperparameter optimization written in Python, minimizing the $95th$ percentile of the error loss function for 100 evaluation steps on the following hyperparameters: *max_depth*, *learning_rate*, *n_estimators*, *reg_alpha*, *reg_lambda*, *gamma*, *subsample*, *colsample_bytree*, and *min_child_weight*. The resulting values are listed in Table 4.2. With the hyperparameters appropriately tuned, the model was trained on the entire training set, and then evaluated on the test set.

The outcomes shown in Figure 4.8 suggest that excluding high ϵ_{95} values, XGBoost fails to outperform the baseline approaches. This indicates that, in our case, increasing the complexity of the model does not does not positively influence its accuracy. Hence, it makes sense to investigate different approaches able to better exploit the environmental information contained within the dataset.

Graph networks approach

In this specific section, we propose and evaluate the effectiveness of a transductive graph neural network approach which, unlike the inductive approaches analysed

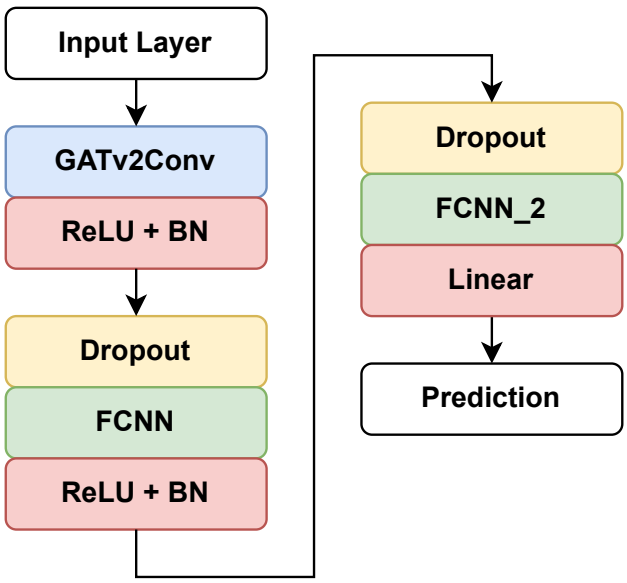


Figure 4.9: Architecture of the graph neural network model considered for the temperature prediction task.

so far, is able to learn dependencies and interactions between the elements within the simulated room. The GNN solution, formerly proposed in Section 3.2.3, is oriented towards the selection, based on attention mechanisms, of informative sensor nodes to be used in the predictive task. For this predictive task, instead, a more extensive architecture able to exploit knowledge coming from a graph containing nodes related to each element placed inside the monitored environment is needed. On this premise, as depicted in Figure 4.9, our proposed solution is obtained by increasing the capacity of the employed GAT layer and adding a couple of dense fully connected layers on top of it.

Considering the task of indoor temperatures prediction, given an input graph, the GAT layer allows to create one embedding for each node. Such embeddings are obtained reducing information, like temperatures, air flows exchanges or radiation effects coming from a node neighborhood. Finally, the fully connected layers compute an embedding representing a node's state information from which the temperature value is predicted as outcome.

Beside comparing the accuracy attained in temperature prediction with respect to other approaches, this experimentation aims to assess the impact, in terms of improved accuracy, of incorporating information from various sets of elements characterizing the indoor environment. For this purpose, the simulation data have been modeled as four set of graphs which differ in terms of node types and features employed. In detail, the four sets of graphs considered for this experimentation are the

following:

- *sensors only* graphs, $\mathcal{G}_{sens} = \{G^t | G^t = (V^t_{sens}, E_{sens})\}_{t \in T}$ where T is the set of time samples considered, $V_{sens} = V_{refs} \cup V_{test}$ the set of all provided room sensors, and E_{sens} the set of edges defined as $\{(v_i, v_j) | v_i, v_j \in V_{sens}$ and $v_i \neq v_j\}$; at each timestep $t \in T$, there corresponds a different graph, characterized by the observed values contained in the set of node features $\mathbf{h_v} = \{\mathbf{h}_i | v_i \in V_{sens}\}$ with $\mathbf{h_i} \in \mathbb{R}$ air temperatures in Celsius (°C), along with the set of time-independent edge features $\mathbf{h_e} = \{\mathbf{h}_{i,j} | (v_i, v_j) \in E_{sens}\}, \mathbf{h}_{i,j} \in \mathbb{R}^3$ where the 3 real values are the coordinates differences on the z, y, x axes between the starting node and the destination node of the associated edge $(v_i, v_j) \in E_{sens}$;

- *room nodes only* graphs, $\mathcal{G}_{room} = \{G^t | G^t = (V^t_{room} \cup V^t_{test}, E_{room})\}_{t \in T}$ where T is the set of time samples considered, V_{room} the set of room nodes modeling walls and objects located inside the room, V_{test} the set of test sensors, and E_{room} the set of edges defined as $\{(v_i, v_j) | v_i, v_j \in V_{room} \cup V_{test}$ and $v_i \neq v_j\}$; here the set of node features is defined as $\mathbf{h_v} = \{\mathbf{h}_i | v_i \in V_{room} \cup v_{test}\}$ with $\mathbf{h_i} \in \mathbb{R}^3 \times \mathbb{B}^{12}$ where the 3 real input features associated to each node v_i are the emitted air temperatures in Celsius (°C), the emitted mass flow in kilogram per second (Kg/s) and the node orientation angle with respect to the z axis expressed in radians, and the 12 Boolean features are the one-hot encoding of the node type which could be one choice from all possible room element types $\mathcal{T}$; the set of time-independent edge features is $\mathbf{h_e} = \{\mathbf{h}_{i,j} | (v_i, v_j) \in E_{room}\}$, with $\mathbf{h}_{i,j} \in \mathbb{R}^3$ defined as above;

- *all nodes* graphs, $\mathcal{G}_{all} = \{G^t | G^t = (V^t_{refs} \cup V^t_{room} \cup V^t_{test}, E_{room})\}_{t \in T}$ where T is the set of time samples considered, V_{refs} the set of room reference sensors, V_{room} the set of room nodes, V_{test} the set of test sensors, and E_{all} is the set of edges defined as $\{(v_i, v_j) | v_i, v_j \in V_{refs} \cup V_{room} \cup V_{test}$ and $v_i \neq v_j\}$; the set of node features is defined as $\mathbf{h_v} = \{\mathbf{h}_i | v_i \in V_{refs} \cup V_{room} \cup v_{test}\}$ with $\mathbf{h_i} \in \mathbb{R}^3 \times \mathbb{B}^{12}$ as above; also for the set of time-independent edge features $\mathbf{h_e} = \{\mathbf{h}_{i,j} | (v_i, v_j) \in E_{all}\}$, $\mathbf{h}_{i,j} \in \mathbb{R}^3$ is defined as above;

- *no edge features* graphs, $\mathcal{G}_{noedge} = \{G^t | G^t = (V^t_{refs} \cup V^t_{room} \cup V^t_{test}, E_{room})\}_{t \in T}$ defined as the *all nodes* graphs set, but without edge features (i.e., with $\mathbf{h_e}$ empty).

For this experimentation, the evaluated GNN has been implemented extending the GATv2Conv model from the PyTorch Geometric library as follow:

- the embedding $\mathbf{h}'_i$ for node v_i is computed through a 2 head GAT layer (GATv2Conv on the upper left side of Figure 4.9) consisting of 128 units, aggregating information of its neighbors as in the equation

$$\mathbf{h}'_i = ReLU\left(\sum_{v_j \in \mathcal{N}_i} \alpha^{(1)}_{i,j} \mathbf{W}^{(1)} \mathbf{h}_j \| \sum_{v_j \in \mathcal{N}_i} \alpha^{(2)}_{i,j} \mathbf{W}^{(2)} \mathbf{h}_j\right),$$

  where $\mathbf{W}^{(1)}$ and $\mathbf{W}^{(2)} \in \mathbb{R}^{128 \times F_v}$, are learnable linear transformation weight matrices, $\mathbf{h}_j$ are the input features associated to a node v_j, F_v is the number of features associated to each node, $\alpha^{(1)}_{i,j}$ and $\alpha^{(2)}_{i,j} \in \mathbb{R}$ are the attention coefficients for nodes $v_i, v_j \in V$ of, respectively, the first and the second attention head. The output of this step is obtained passing the vector $\mathbf{h}'_i \in \mathbb{R}^{256}$ to a BatchNormalization layer;

- after a Dropout layer with 0.5 probability, the resulting vector in $\mathbb{R}^{256}$ is passed to a dense fully connected layer (FCNN on the lower left side of Figure 4.9) consisting of 256 units with a ReLU activation function followed by a BatchNormalization layer;

- as final step, again after a Dropout layer with 0.5 probability, the resulting vector is passed to a dense fully connected layer (FCNN_2 on the right side of Figure 4.9) of 256 units. At last, on the top of this architecture a final linear layer with one output unit is applied.

Training the newly defined GNN on the four sets of graphs $\mathcal{G}_{sens}, \mathcal{G}_{room}, \mathcal{G}_{all}$ and $\mathcal{G}_{noedge}$, we obtain the respective models $GNN_{sens}, GNN_{room}, GNN_{all}$ and GNN_{noedge}. This phase is performed through the Adam optimization algorithm with *learning_rate* equal to 0.001 minimizing the *mean_squared_error* loss function based on the temperature values actually observed and those predicted by the GNN models for each sensor in V_{test} and timestep $t \in T_{train}$. Furthermore, the temperature values for sensor nodes in V_{test} are masked and each feature of the embeddings in $\mathbf{h_v}$ and $\mathbf{h_e}$ is standardized considering values observed in all of the T_{train} samples. Additional hyperparameters values adopted for the training phase are the *batch_size* set to 16, the momentum and RMSProp terms $\beta_1 = 0.9$ and $\beta_2 = 0.999$, and the training *epochs* equal to 500. Finally, the evaluation of the trained model is carried out for all graphs G^t with t in T_{test} where, again, the temperature values to predict for all nodes in V_{test} are masked. The architecture of the model and all the other hyperparameters were chosen through iterative random search tuning performed on a fixed 80–20% training-validation subsplit of timesteps in T_{train}.

From the results shown in Figure 4.8 it is clear that the proposed model based on graph networks is able to outperform all other approaches as regards the smallest, the median and the largest values of the $95th$ percentile of the prediction error. Although the GNN_{noedge} model (without edge features) behaves similarly to XGBoost, the inclusion of edges features (models GNN_{sens}, GNN_{room} and GNN_{all}), and of nodes related to the elements of the room in addition to sensor nodes (as for model GNN_{all}), allows to consistently increase the accuracy of the model.

4.2.2 Analysis with different room conditions

Apart from the ability of yielding accurate predictions, a desired feature of the methodologies under study is that of being adaptable to environments with different simulated conditions and configurations, in some terms, proving to have learned physical characteristics and phenomena caused by the presence of certain elements within a room. In the remainder of this section, we evaluate both the adaptability of the sensor placement method considering a simulated environment with a different internal configuration, and the behavior of the predictive model in case of mixed environmental conditions.

Sensor placement method analysis

With the aim of evaluating the relocation capability of the solution based on graph attention mechanisms proposed in Section 3.2.2, we define an algorithm able of transferring the outcome of this procedure to a different (in terms of configuration of elements placed inside) room simulation.

This latter is outlined in Algorithm 1 and takes as inputs the set of selected sensors V_{refs}, the set of nodes in the original room V_{room}, the scores $score_i$ yielded by the sensor placement procedure for each $v_i \in V_{sens}$ and the set of nodes in the new room V'_{room}. At the beginning, for each node type $\tau \in \mathcal{T}$ the support set $\mathcal{S}_\tau$ is initialized as $\emptyset$ (line 1). These support sets are used to trace the information necessary to arrange a set of sensors in the new room considering the type of the nodes placed in it. On lines 2–11 this support information is extracted, where *type* is a function $V \to \mathcal{T}$ returning the type of a node, *coords* a function $V \to \mathbb{R}^3$ returning the spatial coordinates of a node and *angle* a function $V \to \mathbb{R}$ returning the orientation angle of a node. In detail, for each sensor $v_i \in V_{refs}$ and for each node $v_j \in V_{room}$ of type $\tau \in \mathcal{T}$ such that the Euclidean distance between v_i and v_j is less than or equal to d_{max}, a tuple in $\mathbb{R}^3 \times \mathbb{R} \times \mathbb{R}$ is added to $\mathcal{S}_\tau$. In detail, this tuple contains the vector given by the element-wise difference of the coordinates of nodes v_i and v_j, the orientation angle θ_j of v_j and the score $score_i$ associated with

v_i.

Algorithm 1: Room Sensors Transfer

input : selected sensors set V_{refs}, sensor score $score_i$ $\forall v_i \in V_{sens}$, set of room nodes V_{room}, set of new room nodes V'_{room}

global: max distance d_{max}, node type set $\mathcal{T}$

1: **init** $\mathcal{S}_\tau \leftarrow \emptyset, \forall \tau \in \mathcal{T}$
2: **for** $v_i \in V_{refs}$ **do**
3: $c_i \leftarrow coords(v_i)$
4: **for** $v_j \in V_{room}$ **do**
5: $c_j, \tau \leftarrow coords(v_j), type(v_j)$
6: **if** $dist(c_i, c_j) \leq d_{max}$ **then**
7: $\theta_j \leftarrow angle(v_j)$
8: $\mathcal{S}_\tau \leftarrow \mathcal{S}_\tau \cup \{\langle c_i - c_j, \theta_j, score_i \rangle\}$
9: **end if**
10: **end for**
11: **end for**
12: **for** $\langle c, \theta, score \rangle \in \mathcal{S}_\tau, \forall \tau \in \mathcal{T}$ **do**
13: **for** $v_k \in V'_{room}$ such that $type(v_k) = \tau$ **do**
14: $c', \theta' \leftarrow coords(v_k), angle(v_k)$
15: $c' \leftarrow (c \cdot \mathcal{R}_{\theta' - \theta}) + c'$
16: **if** $score'_{c'}$ is defined **then**
17: $score'_{c'} \leftarrow score'_{c'} + score$
18: **else**
19: $score'_{c'} \leftarrow score$
20: **end if**
21: **end for**
22: **end for**
23: **return** $score'$

Thereafter, following the steps on lines 12–24, the sensors selected for the original room are mapped to the new one through a new score variable $score'_c$ defined as a map $\mathbb{R}^3 \rightarrow \mathbb{R}$ which associates to a point c in the new room $\mathbb{R}^3$ space a score representing the importance of that position if considered as a predictor sensor. More in detail, for each tuple $\langle c, \theta, score \rangle$ in the support $\mathcal{S}_\tau$ and each new node v_k in V'_{room} with $type(v_k)$ equal to τ, the coordinates c are transferred in the new room space considering the position and orientation angle of v_k. The salient point of this strategy is found in line 16, where the spatial coordinates vector c_k of node v_k is summed element-wise to the dot product of c and a rotation matrix defined as

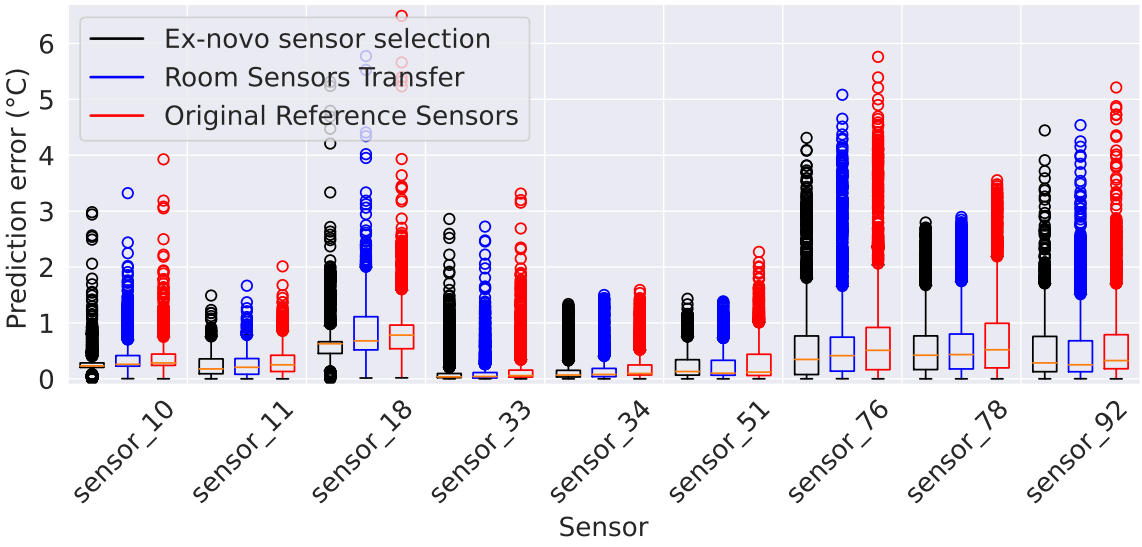


Figure 4.10: Boxplots of sensor prediction errors provided in the new room configuration 2 considering as reference sensors the outcome of an ex-novo graph attention-based selection and the room sensors transfer procedure compared to the original reference sensors selected for room configuration 1.

$$\mathcal{R}_{\theta'-\theta} := \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta' - \theta) & \sin(\theta' - \theta) \\ 0 & -\sin(\theta' - \theta) & \cos(\theta' - \theta) \end{bmatrix}$$

where θ' is the orientation angle of v_k and θ the angle stored in the support tuple. This makes it possible to reallocate reference sensors in the new room considering the orientation angle of the elements in the original setting (θ) coupled to that of the elements in the new configuration (θ'). Finally, exploiting the output of Algorithm 1 we obtain the set of new sensor nodes $V' = \{c' \mid score'_{c'}$ is not undefined and $score'_{c'} > 0\}$. The final set of reference sensors is selected sorting the values of $score'_{c'} \forall c \in V'$ in decreasing order and applying to the resulting curve the Kneedle algorithm (as already illustrated in Section 3.2.2).

As for the experimentation on our case study, we applied the Algorithm 1 to the outcome of the sensor placement procedure presented in Section 3.2.4 trained on the sequence of events Seq_w data for room configuration 1 (so that $V_{refs} = V^{att}_{refs}$) with the aim of obtaining an optimal sensor arrangement for room configuration 2 (see Figure 3.16). Considering a value d_{max} equal to $\sqrt{3}$ m, the set of sensors obtained after the transfer procedure is $V'_{refs} = \{66, 65, 41, 29, 61, 62, 53, 30, 57\}$. For the sake of comparison, we perform an ex-novo execution of the graph attention-based sensor allocation procedure for room configuration 2 obtaining as outcome the set

of selected sensors $V'^{att}_{refs} = \{66, 65, 72, 80, 41, 29, 61\}$. In Figure 4.10 a comparison of the obtained results using the three sets of reference sensors V'^{att}_{refs} (ex-novo graph attention-based selected sensors for room configuration 2), V'_{refs} (outcome of the room sensors transfer procedure) and V_{refs} (original reference sensors selected for room configuration 1) as predictors for the temperatures observed by nodes in V_{test} is reported. Here, a graph attention neural network configured and trained as outlined in Section 4.2.1 is employed. As can be seen, for the models trained considering nodes in V'^{att}_{refs} and V'_{refs} as reference sensors, the error distributions are substantially similar, especially as regards intermediate values. Moreover, the model trained considering sensors in V_{refs} gives the worst performances for both intermediate error values and outliers.

To conclude, it is worth noticing that, in our case, having a discrete grid of sensing monitors the values of $score'$ (lines 16–20) are assigned to sensors located within a radius of $\sqrt{3}$ m from the estimated coordinates c'. However, the proposed procedure is meant to be more general, providing transfer solutions in a potentially continuous $\mathbb{R}^3$ space.

Predictive model analysis

Here we evaluate the graph network-based solution presented in Section 4.2.1 with reference to a different indoor scenario. First of all, the training of the predictive model is performed as defined in Section 4.2.1 on data obtained from the independent simulation of each individual event included in the winter and summer sequences (Seq_w and Seq_s) described in Section 3.2.1 (exception made for event 4, regarding the activation of the air conditioning vents) for room configuration 1. In this step, the sensors in V^{att}_{refs} are used as predictors, and the remaining ones in $V \setminus V_{test}$ as labels. Subsequently, the evaluation of this pre-trained GNN model takes place on the unseen simulated sequences of Seq_w and Seq_s, as above for sensors in $V \setminus V^{att}_{refs}$ in room configuration 1. In details, Seq_w and Seq_s differ from the training data as, being simulated in sequence, each event depends on the final environmental conditions of the preceding ones.

Below, the graph network model trained on independent events is compared with 3 other models with the same architecture: one trained only on Seq_w, one only on Seq_s, and one on both Seq_w and Seq_s. The split between training and test set was performed extracting T_{train} and T_{test} which contain, respectively, 80% and 20% of the time instants of each event in Seq_w and Seq_s.

The results in Figure 4.11 and 4.12 show the distribution of the $95th$ percentile of the prediction error ϵ_{95} for sensors in V_{test} regarding each event in the evaluation sequences of, respectively, Seq_w and Seq_s (considering samples in T_{test}). As regards

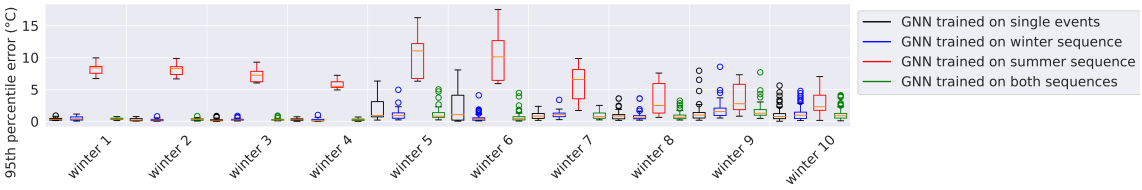


Figure 4.11: Boxplots of the $95th$ percentile of the error provided for the considered events in the winter sequence Seq_w by GNNs trained on different conditions.

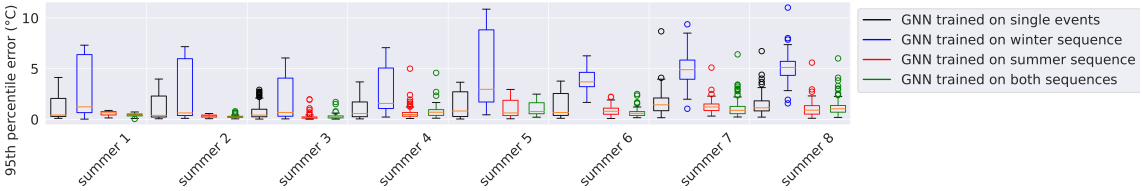


Figure 4.12: Boxplots of the $95th$ percentile of the error provided for the considered events in the summer sequence Seq_s by GNNs trained on different conditions.

the median value of ϵ_{95}, the model trained on single events provided comparable performances both to the model trained ad hoc for the sequence Seq_w, and to that trained on Seq_s. The same holds for the model trained on both the Seq_w and Seq_s sequences, with lower maximum values than the model trained on single events. As might be expected, GNN trained on Seq_w performed poorly on Seq_s events, and vice versa. In particular, the maximum values of ϵ_{95} are very high using a Seq_s-trained model to predict events in Seq_w. This is probably determined by the greater effect yielded in winter by heaters compared to the heat dispersion supplied by them in summer.

4.2.3 Discussion

In this section, we presented a novel framework for the efficient thermal monitoring in a general indoor environment. Our proposal encompasses multiple, relevant aspects of virtual sensing, including the placement of physical sensors, the choice of the predictive model, and the evaluation of its performance.

For the sake of clarity, below we summarize how the steps of the proposed procedure are organized. The overall idea is to carry out simulations from sufficiently detailed and complex 3D environment models, in order to cover all types of furniture elements, events and physical phenomena of interest (e.g., simulating the switching on of a radiator or the opening of a window). Starting from this simulated data, the sensor placement procedure described in Section 3.2.2 returns a list of reference positions whose observed temperature values allow the training of a predictive

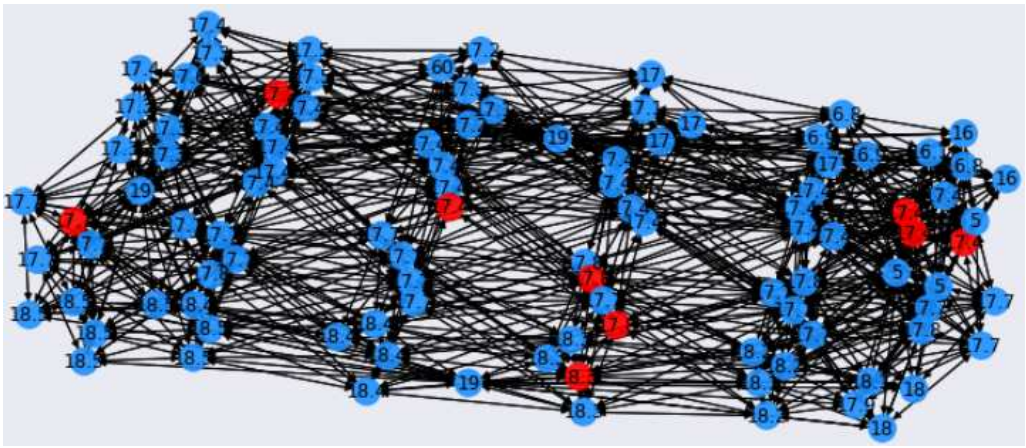


Figure 4.13: Example of input graph structure.

model acting as a simulation approximator. This is accomplished by encoding the environmental information into a graph structure serving as input during the training process, as outlined in Section 4.2.1. Figure 4.13 shows an example of such a structure where blue nodes represent reference sensors or room nodes, and red nodes virtual sensors. The values associated with them correspond to temperature values. The adoption of this graph structure has several advantages. In fact, the obtained models are able to work independently of the order of nodes and capture dependencies and interactions between them.

Thus, the graph network trained on the initial simulation data can be employed with good efficacy for temperature monitoring in similar contexts, regardless of its environmental conditions, or usage scenarios. This is possible provided that the initial simulations include all types of elements placed in the new configurations and all the events capable of generating the physical phenomena to be monitored modeled with sufficient complexity and level of detail.

Operationally, it suffices to prepare an input graph for the model with nodes of correct typology and relative features appropriately set according to what is actually present in the new room configuration to be monitored. Once the set of nodes has been encoded, the coordinates of the reference sensors to be placed in the room are obtained using the algorithm in Section 4.2.2. After an inital training phase on simulation data as described in Section 4.2.1, when the physical sensors have been arranged and their observed temperatures have been collected, an input graph for the predictive model is created. At this point, to predict the temperature in a specific point of the room, a sensor type node with the desired coordinates is employed. Spatial interpolation can be easily obtained varying the coordinates of this virtual sensor, by reading the relative output predicted by the graph network per-node model.

5

Monitoring

The ability of sensor systems to generate large volumes of dynamic and heterogeneous data has opened up new possibilities for a wide range of applications, from industrial process control to environmental monitoring. In the previous chapters, we have explored various aspects of sensor systems, including sensor selection and virtual sensing of indoor environments. Once a sensor system has been made operational and efficient, it can be used for tasks such as event, anomaly, or failure detection, providing enhanced control functions.

In many application domains, during its operation a system produces several data streams, that may contain valuable telemetry information. This is the case, for instance, with the logs generated by web servers, smart sensors, and industrial machinery. Such data can be used for tasks like predictive maintenance and early failure detection, typically carried out, due to their complexity, by means of machine or deep learning approaches. Despite their success, these methods hardly provide any guarantees over their execution, a problem which is exacerbated by their lack of interpretability, which is an essential requirement in many critical domains, such as, for instance, healthcare and avionics. In those scenarios, formal methods can, in principle, be used to automatically verify software and hardware systems. However, the presence of different operating conditions combined with the complexity of the system components and their interactions make it quite difficult to define in advance all the relevant conditions that must be guaranteed (or avoided) during execution; moreover, the specification of a complete system model against which to check these properties may be simply impossible [106].

To overcome these limitations, various methods, that combine classic exhaustive formal verification techniques with model-based testing and monitoring, have recently been proposed in the literature (see, e.g., [36, 67]). Here, we focus on the latter. *Monitoring* [106] is a runtime verification technique which is receiving more and more attention from the formal verification community. It allows one to detect the fulfilment or violation of a property (usually expressed by a temporal logic formula) by evaluating a single system run, without requiring a model of the system

being considered. This makes it naturally applicable to data streaming scenarios.

In this chapter, we present a novel online system verification framework that combines monitoring with supervised machine learning and can be used for tasks like preemptive failure detection over streams of data. The framework starts its operation by considering a limited set of properties encoding bad behaviors, to be monitored against the system, learnt during a *warmup* phase and/or specified with the help of domain experts. Then, during the *runtime* phase, by means of an iterative refinement process, the framework autonomously discovers new relevant properties, becoming able, over time, to identify undesired behaviors in advance, and with a significantly higher level of detail and coverage than the initial specifications. The process of property discovery and extraction is carried out by means of an original bi-objective evolutionary algorithm.

The distinctive features of the proposed solution are the following:

- the framework poses as a monitoring-based tool to perform preemptive failure detection;
- its operation relies on the seamless and automatic interaction between formal methods and machine learning approaches;
- thanks to its modularity and flexibility, the framework can be adapted to different application domains and contexts; in particular, Signal Temporal Logic (STL) can possibly be replaced by other logical formalisms for property specification;
- interpretability is a distinguishing feature of the framework, as the produced responses can be easily read by domain experts; this allows people to validate the overall behavior of the framework, and to gain insights about the causes that led to a failure;
- the framework works in an online fashion, and it can adapt to changes in the behavior of the system, due, for instance, to updates or upgrades.

The framework has been evaluated against three public datasets, and it has shown to be able of actually predicting in advance system failures. Results are on par with those obtained from other state-of-the-art solutions that, however, suffer from a lack of interpretability. In order to deal with the task of failure detection, we chose to validate our approach on different datasets and use cases compared to those analyzed in the previous chapters. However, as a future direction it would be interesting to extend the framework to address situations where the focus is not solely on failure detection, but also on anomaly, degradation, or event detection (e.g., violations of comfort constraints in the case of indoor temperature monitoring).

The rest of the chapter is organized as follows. Section 5.1 analyses related work. Section 5.2 provides background knowledge about monitoring, STL, and evolutionary algorithms. Section 5.3 describes the implementation of the evolutionary algorithm used in the property extraction phase. Section 5.4 shows how such an algorithm has been incorporated in the proposed framework. The experimental evaluation of the framework is reported in Section 5.5. Finally, Section 5.6 provides a critical assessment of the work done and discusses its strengths and limitations.

5.1 Learning to detect failure

Learning techniques for the real-time detection of undesired behaviors (failures) of complex systems are getting increasingly popular. A significant line of research makes use of machine learning and deep learning, that realize failure detection via black-box models rather than by providing explicit properties capable of characterizing bad behaviors of a system. Despite their lack of interpretability, that makes it difficult to understand and validate the resulting verdicts, these approaches have been employed in several domains due to their effectiveness. For instance, machine learning strategies based on Logistic Regression (LR), Support Vector Machine (SMV), Random Forest (RF), and K-Nearest Neighbours (KNN) applied to the domains of aircraft components post-flight reports, gearbox failures in industrial robots, high-performance computing, and cloud systems are described in [97, 174, 126]. Deep learning solutions are typically exploited to extract temporal relations in time series data, as witnessed in [65, 3, 110, 144], where Long Short-Term Memory (LSTM) and Recurrent Neural Networks (RNNs) are applied to the domains of job failures in large-scale cloud data centers, turbofan engine degradation, hard drive telemetry data, and heart atrial fibrillation detection on routine screening electrocardiogram (ECG) signals. A common limitation of all these solutions is that historical data used for the predictions are often defined through a time window of fixed size, which may be inadequate when heterogeneous failure behaviors have to be captured.

In the attempt to cope with the interpretability requirement, which is fundamental in many critical domains, some approaches for the extraction of properties that distinguish between failure and normal execution traces of a system have been recently proposed in the literature [92, 132, 18, 125, 133, 17, 40]. However, learning temporal properties from the observed system traces is a challenging task that involves intractable optimization problems [92]. To overcome them, heuristics were suggested [92, 132, 18, 125]. In this spirit, some ad hoc, domain-specific solutions have been devised to assess the condition of electrical rotating machines through real-time vibration measurement and analysis instruments [47], to discover temporally-

constrained alarm sets from dynamic systems' logs [56], to diagnose rolling bearings faults through a hardware architecture with a reconfigurable logic based on field programmable gate arrays (FPGAs) [89], to detect system intrusions through temporal logic specifications [131], and to ensure the safety of synthesized policies for robotics through model predictive shielding [5]. Nonetheless, a general technique, applicable to different domains and contexts, is still missing. A step towards such a goal was taken in [29], where an STL-based solution to the problem of detecting ineffective respiratory effort in intensive care patients, which includes a learning phase supporting some adaptive behaviors, is outlined. Still, the generative data models employed in the learning phase are tailor-made, thus limiting the flexibility and generality of the proposed solution.

With the goal of generalizability in mind, approaches explicitly aimed at combining the points of strength of machine learning and formal methods have been recently proposed. Specifically, in [133, 17, 40], techniques for the mining of STL properties that distinguish between two different sets of time series data are presented. The proposal in [133] relies on a genetic algorithm combined with parameter learning through Gaussian process confidence upper bound. The solution originally presented in [18], and then extended with online learning in [17], exploits a decision tree learner based on STL primitives. Finally, the approach in [40] relies on a reinforcement learning-based property extractor that combines data- and knowledge-driven methodologies. Still, the aforementioned proposals significantly differ from what is presented in this work, as they are not designed to work iteratively, managing a pool of properties in real-time. Beyond STL, in [181], a failure prediction method for cloud data centers, based on message pattern recognition via Bayesian probability, is described. As for failure detection in cyber-physical systems, a Ripple down rule-based (RDR) framework was proposed in [91], that exploits a machine learning technique based on the algorithm InductRDR [64]; the result is then maintained by domain experts, who refine the RDR knowledge base.

A related field is that of specification mining, whose goal is to generate/integrate the formal specification of a system by analyzing its execution traces. Various approaches to this problem have been proposed in the literature. In [105], a Linear Temporal Logic (LTL) property template miner, based on support and confidence thresholds, is devised. A Bayesian inference-based probabilistic model that generates LTL task specifications from examples, by exploiting a Markov Chain Monte Carlo algorithm, is outlined in [171]. In [88], an algorithm to infer LTL specifications by combining the representational power and interpretability of temporal logic with the generalizability of inverse reinforcement learning is proposed. The problem of mining finite state automata to generate formal specifications in the context of software applications and libraries is dealt with in [103]. To this end,

the authors make use of prefix tree acceptors, language models based on recurrent neural networks, and clustering algorithms to merge similar automaton states. In the context of failure detection and specification mining, it is worth mentioning that recent developments in learning temporal formulas for sensor-based prediction, such as temporal decision trees [27] and forests [119], provide an alternative approach that is focused on leveraging temporal relationships in time series data. These approaches hold potential for enhancing both the effectiveness and interpretability of failure detection systems. However, it is important to consider certain drawbacks. The construction and training of temporal decision trees and forests can be computationally intensive, particularly when dealing with large-scale and high-dimensional datasets. Additionally, the scalability of these methods might be a concern when confronted with substantial amounts of streaming data or when the dynamics of the system change over time. To conclude, despite the relevance of specification mining, the proposed solutions extract non-contrastive properties from the observed executions of a system, and thus they cannot be naturally applied to failure detection, where properties able to discriminate between good and bad behaviors are needed.

5.2 Background knowledge

In this section, we recall some basic notions about monitoring and STL

5.2.1 Monitoring

As introduced in Section 2.5, while classic verification techniques, like, for instance, model checking [46], perform an exhaustive analysis of the behaviors of a system, monitoring [106] aims at establishing satisfaction or violation of a property by analyzing a finite prefix of a single behavior (*trace*/*run*), and then issuing an irrevocable verdict [106]. It is thus a lightweight technique, but the gain in efficiency is paid in terms of expressivity: monitorable properties are a subset of those expressible in temporal logics commonly used for automated verification.

We say that a property is *positively* (resp., *negatively*) *monitorable* if every trace satisfying (resp., violating) it features a finite prefix that witnesses the satisfaction (resp., violation). A *monitorable* property is a property that is either positively or negatively monitorable. Safety properties, informally requiring that "*something bad will never happen*", are negatively monitorable, as their violation is witnessed by a finite prefix exhibiting a violation; dually, co-safety properties, stating that "*something required will eventually happen*", are positively monitorable. Notably, there are meaningful properties, like, e.g., "*a good state is accessed infinitely often*"

(an essential ingredient of strong fairness requirements [118]), which are clearly neither positively nor negatively monitorable.

As we will see in Section 5.4, the online nature of monitoring makes it a natural candidate for the proposed framework.

5.2.2 Signal Temporal Logic (STL)

Signal Temporal Logic (STL) [116] extends propositional logic with future modalities that allow one to express temporal properties over linear structures. It can be directly applied to time series data characterized by continuous values.

Let $\mathbb{N}_{>0}$ (resp., $\mathbb{N}_{[t,t']}$) be the set of positive naturals (resp., naturals in between t and t', for all $t, t' \in \mathbb{N}$) and $\mathbb{R}^n$ be the n-dimensional Euclidean space over reals. A discrete-time STL *signal* (or *trace*) is a function $x : \mathbb{N} \to \mathbb{R}^n$, for some $n \in \mathbb{N}_{>0}$; a *partial signal* is a function $x : \mathbb{N}_{[0,t]} \to \mathbb{R}^n$, for some $n \in \mathbb{N}_{>0}$ and $t \in \mathbb{N}$.[1] We denote the *length* of a (partial) signal x by $len(x)$. For a signal x, it holds that $len(x) = \infty$, whereas $len(x) = t + 1$ for a partial signal $x : \mathbb{N}_{[0,t]} \to \mathbb{R}^n$. Let $\mathcal{X}$ (resp., $\bar{\mathcal{X}}$) be the set of signals (resp., partial signals). If the codomain of a (partial) signal x is $\mathbb{R}^n$, then n is the *dimension* of x, denoted by $|x|$. Let $x \in \mathcal{X} \cup \bar{\mathcal{X}}$. We denote by x_i, with $1 \leq i \leq |x|$, the function from the domain of x to $\mathbb{R}$ such that $x_i(t)$ is equal to the i-th component of $x(t)$, for all t. Moreover, we denote by $x[j, k]$, with $0 \leq j \leq k < len(x)$, the restriction of the function x to the domain $\mathbb{N}_{[j,k]}$.

The syntax of STL is given by the grammar:

$$\phi ::= \top \mid x_i \geq c \mid \neg\phi_1 \mid \phi_1 \wedge \phi_2 \mid \phi_1 \mathsf{U}_I \phi_2,$$

where x is a signal, $1 \leq i \leq |x|$, $c \in \mathbb{R}$, and I is an interval of the form (a, b), $(a, b]$, $[a, b)$, or $[a, b]$, with $a \in \mathbb{N}$, $b \in \mathbb{N} \cup \{\infty\}$, and $a \leq b$. Modality U (*until*) is paired with an interval I which defines its validity scope. For every $t \in \mathbb{N}$ and interval $I = (a, b)$, we denote by $t + I$ the interval $(t + a, t + b)$ (the same for intervals of the forms $(a, b]$, $[a, b)$, and $[a, b]$). Derived modalities are defined as usual. As an example, modalities *eventually* and *globally* are defined as $\mathsf{F}_I \phi = \top \mathsf{U}_I \phi$ and $\mathsf{G}_I \phi = \neg \mathsf{F}_I \neg \phi$, respectively.

STL pairs the standard Boolean semantics with a quantitative one, which measures the *robustness* of the satisfaction of ϕ by a signal x at a time $t \in \mathbb{N}$.

The quantitative semantics of STL is inductively defined as follows:

- $\rho(\top, x, t) = +\infty$;

[1] As a matter of fact, STL allows one to deal with continuous-time signals by simply redefining x as $x : \mathbb{R}_{\geq 0} \to \mathbb{R}^n$, where $\mathbb{R}_{\geq 0}$ is the set of non-negative reals. Here, we restrict ourselves to the discrete-time case given that a sampling is required to represent time series data within a dataset.

- $\rho(x_i \geq c, x, t) = x_i(t) - c$;
- $\rho(\neg\phi, x, t) = -\rho(\phi, x, t)$;
- $\rho(\phi_1 \wedge \phi_2, x, t) = \min\{\rho(\phi_1, x, t),\ \rho(\phi_2, x, t)\}$;
- $\rho(\phi_1 \mathsf{U}_I \phi_2, x, t) = \max_{t_1 \in t+I} \min\{\rho(\phi_2,\ x, t_1),$ $\min_{t_2 \in [t,t_1)} \rho(\phi_1, x, t_2)\}$.

The Boolean semantics of STL is defined on the basis of the sign of $\rho(\phi, x, t)$:

$$x, t \models \phi \text{ if and only if } \rho(\phi, x, t) \geq 0.$$

Finally, given a partial signal $x \in \bar{\mathcal{X}}$ over $\mathbb{N}_{[0,t]}$, the set of *completions* of x is defined as $\mathcal{C}(x) = \{\hat{x} \in \mathcal{X} \mid \hat{x}(t') = x(t') \text{ for all } t' \in \mathbb{N}_{[0,t]}\}$.

STL monitoring is formally defined by the function $mon : \bar{\mathcal{X}} \times \mathsf{STL} \to \{\top, \bot, ?\}$ such that $mon(x, \varphi)$ returns $\top$ iff $\hat{x}, 0 \models \varphi$ for all $\hat{x} \in \mathcal{C}(x)$ iff $\rho(\varphi, \hat{x}, 0) \geq 0$ for all $\hat{x} \in \mathcal{C}(x)$; $\bot$ iff $\hat{x}, 0 \not\models \varphi$ for all $\hat{x} \in \mathcal{C}(x)$ iff $\rho(\varphi, \hat{x}, 0) < 0$ for all $\hat{x} \in \mathcal{C}(x)$; ? otherwise.

The choice of STL as the formalism for the specification of the properties to monitor has various advantages. First of all, STL allows one to directly deal with real values, still featuring quite compact and interpretable formulas. Moreover, its quantitative semantics provides one with an additional tool to evaluate the behavior of the extracted formulas, a feature that will be described in detail in Section 5.3.

5.2.3 Monitoring bounded STL formulas

Monitoring properties that refer to both the current and the future behavior of a system is a challenging task since their evaluation at a given time t may also depend on the observed inputs at some time $t' > t$. In Section 5.2.1 and Section 5.2.2, we introduced the notion of monitorable properties and provided syntax and semantics of STL, respectively. To the best of our knowledge, no tool supporting the monitoring of arbitrary STL formulas is available. Luckily, in most application domains, the properties to monitor can be expressed by means of bounded-time STL (bSTL) formulas, and a tool to deal with such a class of formulas has been developed in [136]. Basically, the fragment bSTL constrains the interval I associated with modality U to be finite, that is, $I = [a, b]$, with both a and b belonging to $\mathbb{N}$ ($b = \infty$ is excluded).

Let ϕ be a bSTL formula. By analyzing its syntactic structure, one can compute a temporal *horizon* $H(\phi)$, that intuitively represents the maximum number of (future) time points that one must take into consideration to establish whether or not ϕ is true. In the following, when evaluating a bSTL formula ϕ, with temporal horizon

$H(\phi)$, at a given time t, the monitor will wait until time $t + H(\phi)$ is reached, since at that time all the data necessary for the quantitative evaluation of the formula and the possible formulation of a $\top$ or $\bot$ verdict have surely been observed. As an example, the horizon of the bSTL formula $\phi = x \geq 3\ \mathsf{U}_{[0,3]}\ x \geq 5$ is 3, and thus only after 3 time units we can complete its (quantitative) evaluation.

Formally, the temporal *horizon* $H(\phi)$ of a bSTL formula ϕ is defined as follows:

- $H(\top) = 0$;
- $H(x_i \geq c) = 0$;
- $H(\neg\phi) = H(\phi)$;
- $H(\phi_1 \wedge \phi_2) = max\{H(\phi_1), H(\phi_2)\}$;
- $H(\phi_1\ \mathsf{U}_{[a,b]}\phi_2) = b + max\{H(\phi_1) - 1, H(\phi_2)\}$.

Notice that the monitor, when applied to a bSTL formula ϕ, may output the truth values $\top$ or $\bot$, as well as the undefined verdict ? when the horizon of ϕ still has to be reached. As previously mentioned, a $\top$ or $\bot$ can always be reached when the horizon is met.

Formally, bSTL monitoring is defined by the function $b\text{-}mon : \bar{\mathcal{X}} \times \mathsf{bSTL} \rightarrow \{\top, \bot, ?\}$ such that $b\text{-}mon(x, \varphi)$ returns $\top$ if $mon(x, \varphi)$ returns $\top$ and $len(x) \geq H(\varphi) + 1$; $\bot$ if $mon(x, \varphi)$ returns $\bot$ and $len(x) \geq H(\varphi) + 1$; ? otherwise.

Now, we observe that, by the definition of monitoring and the nature of bSTL formulas, monitors evaluate bSTL formulas only based on prefixes of signals of bounded length, the bound depending on the temporal horizon of the formulas. As an example, formula $\varphi = \mathsf{F}_{[0,3]}\ temperature \geq 3$ states that there must exist at least one time point where $temperature \geq 3$ among the first 4 (i.e., $H(\varphi) + 1$) time points of the signal, that is, in the set of time points $\{0, 1, 2, 3\}$. This limits the applicability of monitoring in real-world scenarios where one is interested in detecting the possible occurrence of a given condition at any time point of a signal. This is the case, for instance, with the property: *"in 25 time units from now the temperature will exceed 30 degrees"*. To accommodate for that, we extend the notion of monitoring by making it possible to apply it to any time point, that is, to any suffix of a signal. To this end, building upon function *b-mon*, we define function $eb\text{-}mon : \bar{\mathcal{X}} \times \mathsf{bSTL} \rightarrow \{\top, \bot, ?\}$ such that:

$$eb\text{-}mon(x, \varphi) = \begin{cases} ? & \text{if } len(x) < H(\varphi) + 1 \\ \bigvee_{0 \leq i \leq len(x)-1-H(\varphi)} b\text{-}mon(x[i, len(x) - 1], \varphi) & \\ & \text{otherwise} \end{cases}$$

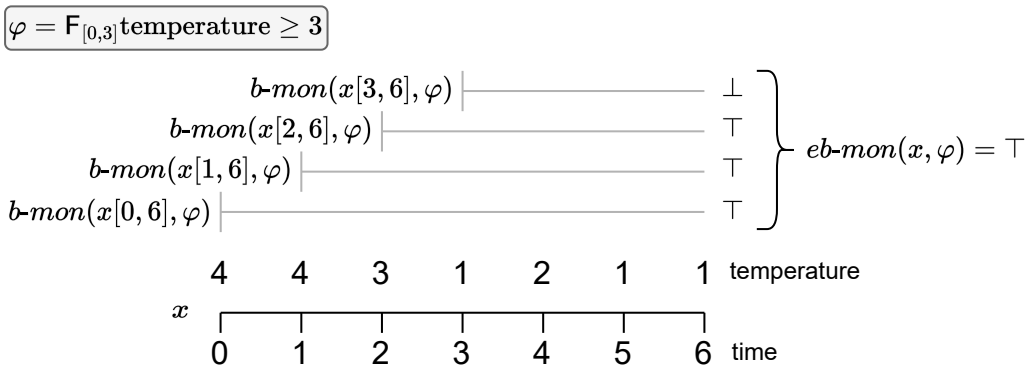


Figure 5.1: *eb-mon* run on a partial signal x. The execution of *b-mon* over all the suffixes of x longer than the horizon is also reported.

We are interested in identifying signals exhibiting bad behaviors, which are encoded by means of bSTL properties. As noted before, a conclusive verdict can be issued for a signal only if it is longer than the horizon of the bSTL property under consideration. Therefore, given a partial signal x and a bSTL property φ, function $eb\text{-}mon(x, \varphi)$ returns ? whenever x is no longer than the horizon of φ. Otherwise, we monitor (through *b-mon*) φ against all suffixes y of x longer $H(\varphi)$: if at least one such monitoring procedures returns $\top$, then so does $eb\text{-}mon(x, \varphi)$, meaning that the signal is considered a bad-behaving one (see Figure 5.1).

The monitoring tool we used to realize the above-described approach is `rtamt`, a Python library for monitoring discrete- and dense-time bSTL properties [136].

5.3 The evolutionary algorithm

As explained in Section 2.4, *evolutionary algorithms* (EAs) are population-based metaheuristics inspired by the process of biological evolution and genetics, that excel in the solution of combinatorial optimization problems.

In this study, we deal with a specific kind of optimization task, that is, *genetic programming*. As already discussed in Section 2.4.1, such a technique evolves programs starting from a population of random solutions. Each individual is encoded by means of a computation tree, where each leaf represents an input value (either a variable or a constant) and internal nodes encode operators.

As we will see, our proposed solution relies on a multi-objective evolutionary algorithm for the property extraction task. Such an EA is able to simultaneously follow different optimization goals, producing a set of Pareto-optimal solutions as a result which could be subsequently combined. Furthermore, it is a flexible approach,

as it allows one to customize the syntax of the generated formulas by constraining the computation trees, e.g., enabling/disabling specific operators or allowing only some kinds of combinations among them.

The evolutionary algorithm we are going to exploit relies on DEAP (Distributed Evolutionary Algorithms in Python) [61], a framework that provides practical tools for the prototyping of custom evolutionary algorithms. In this section, we will illustrate how the different components of the optimizer have been developed.

The algorithm receives a set of finite traces $\mathcal{X}$ (all of the same length) as input, then, it partitions each trace into a normal behavior prefix and a failure behavior suffix, and, finally, it generates a bSTL formula which is able of discerning between the two cases.

5.3.1 Population and its initialization

Each individual belonging to the population consists of a pair (φ, w), where φ encodes a computation tree representing a syntactically correct bSTL formula and w is its associated *normal behavior window length*.

As for φ, it is generated following DEAP's *genHalfAndHalf* method, which outputs a random tree with a maximum height of 6, as suggested by Koza in his seminal work [98]. More precisely, half the time a tree whose leaves have all the same depth is returned; in the remaining cases, different leaves may have different depths.

The window length w is used to partition each trace $x \in \mathcal{X}$ into a normal behavior prefix of length w and failure suffix of length $len(x) - w$ (see the definition of the fitness function below). Note that, in the generation process of the individual, a formula φ with a horizon $H(\varphi) \geq len(x) - w$ might be obtained. In such a case, the individual is discarded and another one is generated. The process is iterated until a valid individual is obtained.

5.3.2 Nodes of the computation tree

A node of the computation tree may represent a constraint, e.g., $x_i \geq c$, a bSTL formula whose outermost operator is a temporal one, e.g., $\varphi \mathsf{U}_{[a,b]} \psi$, or a Boolean formula like, for instance, $\varphi \vee \psi$, where φ and ψ are bSTL sub-formulas which are represented, in their turn, as trees. A node may also encode the following terminal values: (i) interval bounds of a temporal operator, i.e., $[a, b]$, with $a, b \in \mathbb{N}$ and $a \leq b$, (ii) signal identifiers x_i, with $1 \leq i \leq |x|$, and (iii) constants c occurring in formulas, with $c \in Dom(x_i)$ for some i. All these terminals are implemented by means of DEAP's *EphemeralConstants*. As for the length of the normal behavior window, it is implemented as an *EphemeralConstant* w, with $0 < w < len(x)$, where

$len(x)$ is the length of the traces $x \in \mathcal{X}$ (they are all of the same length).

5.3.3 Fitness function

In order to evaluate an individual of the population, each trace $x \in \mathcal{X}$ is logically partitioned into a good behavior prefix $x[0, w-1]$ and a failure suffix $x[w, len(x)-1]$, following a windowing approach which takes w as the length of the normal behavior window. A bi-objective fitness function is then defined by making use of the `rtamt` monitoring algorithm for bSTL.

Formally, the first objective measures how good a formula φ is in discriminating between the normal behavior prefixes and the failure suffixes. For each trace x and each formula φ, let us define the numerical counterpart of $eb\text{-}mon(x, \varphi)$ as follows:

$$NUM(eb\text{-}mon(x,\varphi)) = \begin{cases} 1 & \text{if } eb\text{-}mon(x,\varphi) = \top, \\ 0 & \text{otherwise.} \end{cases}$$

The first objective measure is defined as follows:

$$Acc(\mathcal{X},\varphi) = \frac{\sum_{x\in\mathcal{X}} 1 - NUM(eb\text{-}mon(x[0, w-1+H(\varphi)], \varphi)) + \sum_{x\in\mathcal{X}} NUM(eb\text{-}mon(x[w, len(x)-1], \varphi))}{2 \cdot |\mathcal{X}|},$$

It is worth noticing that, to maximize $Acc(\mathcal{X}, \varphi)$, a formula φ should evaluate to $\bot$ on the normal behavior prefixes and to $\top$ on the failure suffixes. In this respect, it is very important to be able to evaluate a formula φ to $\top$ or $\bot$ till the last instant of the good behavior prefix of a trace x. To this end, we simply extend the prefix with the first $H(\varphi)$ points taken from the failure suffix. Intuitively, the reason is that, otherwise, there may be some failure patterns beginning on the prefix and ending on the suffix, that would not be captured (? verdict).

The second objective measures the robustness of the formula (normalized in the $[0, 1]$ interval) by means of bSTL quantitative semantics. As a preliminary step, at the beginning of the execution of the genetic algorithm, every signal in $\mathcal{X}$ is normalized in the $[0, 1]$ interval so that ρ ranges between -1 and 1. This step is handled implicitly and it does not alter the constant value c of constraints $x_i \geq c$ in the generated output formula, which are still represented with their raw, non-normalized value.

This second objective is defined as follows:

$$Rob(\mathcal{X}, \varphi) = \frac{2 \cdot |\mathcal{X}| - \sum_{x \in \mathcal{X}} \max_{0 \le i \le w-1} \{\rho(\varphi, x, i)\} + \sum_{x \in \mathcal{X}} \min_{w \le i \le len(x)-1-H(\varphi)} \{\rho(\varphi, x, i)\}}{4 \cdot |\mathcal{X}|}.$$

Since two objectives are taken into consideration, no single best-performing solution can be directly selected from a given population by means of the fitness function. Rather, a *Pareto front* of optimal solutions can be identified, containing all *non-dominated* solutions.[2]

As a final note, observe that including the window length w in each individual allows each bSTL formula to define its own (optimal) way of splitting the traces: we may indeed expect different kinds of failure, captured by different formulas, to be characterized by different temporal extensions.

5.3.4 Crossover

Given two parent solutions, the crossover operation generates two new individuals. As for their computation trees, they are generated by one-point crossover (DEAP's *cxOnePoint*). The operator randomly chooses a node in each individual and exchanges the subtrees rooted at it. To avoid bloat, that is, an excessive increase in mean program size without a corresponding improvement in fitness, we placed a static limit of 17 on the children's height (DEAP's *staticLimit*), following once more a suggestion from Koza [98]. When an invalid (over the height limit) child is generated, it is simply replaced by one of its parents, randomly selected. As for the associated window lengths, they are randomly chosen from the parents. Observe that, in performing the crossover operation, non-valid individuals can be generated concerning the relationship between their horizon and normal behavior window w; given an individual with formula φ, if $H(\varphi) \geq len(x) - w$, we replace it by one of the parents, randomly chosen.

5.3.5 Mutation

As for the mutation operation, two operators have been used among those available in DEAP, each one chosen with uniform probability: *mutNodeReplacement*, that

[2] A set $\mathcal{S}$ of solutions for an n-objective problem with fitness function $f = \langle f_i, \ldots, f_n \rangle$ is said to be *non-dominated* if and only if for each $x \in \mathcal{S}$, there exists no $y \in \mathcal{S}$ such that (i) $f_i(y)$ improves $f_i(x)$ for some i, with $1 \leq i \leq n$, and (ii) for all j, with $1 \leq j \leq n$ and $j \neq i$, $f_j(x)$ does not improve $f_j(y)$.

replaces a randomly chosen node in the individual, and *mutEphemeral*, that changes the value of a single constant used within an individual (including, possibly, the window length). As we did for crossover, in order to control bloat we impose a *staticLimit* constraint equal to 17 to the height of the tree. Moreover, it must be checked whether the resulting individual is valid, with reference to its horizon and window length. If this is not the case, the original individual is returned.

5.3.6 Selection

To promote population diversity, we rely on the elitist selection strategy implemented in NSGA-III [50], based on the concepts of *reference points* and *niche preservation* (we refer the reader to [50] for details).

5.3.7 Termination criteria and extraction of final solutions

Let us now focus on the termination criteria of the algorithm and on the extraction of the final solution. As it is commonly done, we impose an upper bound on the number of generations. In addition, we define an *early stopping* strategy, based on the *hypervolume* measure. According to it, the execution of the algorithm is interrupted when no improvement over the hypervolume is observed for a given number of generations. Intuitively, the hypervolume of a Pareto front measures the volume of the search space, bounded by a given reference point, that is weakly dominated by the points on the Pareto front [33]. The assumption is that populations of heterogeneous and well-performing solutions are characterized by a high hypervolume.

Since the EA provides a Pareto front of optimal individuals (φ, w) as its result, to determine the final solution to output we first filter the last population's front keeping all individuals whose formula φ has an accuracy greater than 0.5, that is, better than a random classifier. Then, among such individuals, we return the formula φ of the individual with the highest hypervolume. If no formula with accuracy greater than 0.5 is present in the final front, we return `null`.

5.3.8 Other hyperparameters

The other hyperparameters used by the EA have been established a-priori through grid search tuning performed over a specifically developed synthetic dataset of binary labelled bSTL traces, with the two classes characterized by a heterogeneous set of formulas. They are as follows: *population size* $= 100$ (tested values $[50, 100, 500, 1000]$); *crossover probability* $= 0.7$ (tested values $[0.5, 0.6, 0.7, 0.8]$); *mutation probability* $= 0.5/\sqrt[2]{num\_gen}$ (tested values $[0.3, 0.4, 0.5, 0.6]$); *max generations* $= 500$ (a rather

conservative upper bound); *hypervolume early stopping* = 25 generations (tested values $[10, 25, 50]$). Note that mutation probability starts rather high to ensure an effective *exploration* of the search space; then, it rapidly decays with the number of generations to foster the *exploitation* of the most promising solutions that have been found. Although we recognize that, in principle, each dataset has a different and optimal set of hyperparameters, as we will see, the above values still provide a solid basis when it comes to the overall framework performance, and can thus be considered default choices.

Another hyperparameter, used by the EA in this specific implementation, based on bSTL and `rtamt`, is *max horizon*, whose meaning is fairly natural. Intuitively, it sets an upper bound on the horizon of the formulas that can be explored within the EA. Enforcing a max horizon h has three effects: first, formulas can capture phenomena that are temporally extended at most $h+1$ time points (in terms of the sampling rate of the considered time series); second, given the way `rtamt` (equivalently, *eb-mon*) works, at run time, when evaluating the truth of a formula, the verdict will be ? for the first h time points; third, it has been experimentally observed that the execution time of `rtamt` grows more than linearly with the size of the horizon of a formula. As we will see in Section 5.5, multiple runs of the framework have been taken into consideration in order to collect statistically relevant data. Thus, to allow for faster experimentation, we set a small value of 20 for *max horizon* on all datasets. Although this might seem restrictive, it still allows us to extract meaningful and well-performing properties. In a general usage scenario, *max horizon* should be set by domain experts considering the previously mentioned three aspects. The impact of the *horizon* length on the framework performance is studied in Section 5.5.3.

5.4 The general framework

In the following, we describe the proposed framework for preemptive failure detection. As already pointed out, it works in an *online* fashion and it uses the `rtamt` monitoring algorithm to check the incoming system trace for undesired behaviors. As we will see, in terms of binary classification, the occurrence of a bad behavior is considered as a *positive* event. Thus, a false positive corresponds to an erroneous indication of a bad situation, while a false negative corresponds to a missed detection. Bad behaviors are encoded by bSTL formulas, which are collected in a monitoring pool $\mathcal{P}$.

Operationally, we distinguish between two distinct execution phases of the framework: an optional *warmup* phase and a *runtime* phase. In the first one, the pool

$\mathcal{P}$ is populated with an initial set of formulas encoding bad behaviors, following a *teacher forcing*-like approach [184] on supervised training data. In the second one, the framework online monitors the system, starting with a non-empty pool $\mathcal{P}$.

Algorithm 2: Framework execution (*warmup* phase)

input: initial pool $\mathcal{P}$ of formulas, training dataset $\mathcal{X}$

1: **for** $(x, l) \in \mathcal{X}$ **do**
2: $has\_triggered \leftarrow \bot$
3: $\mathcal{S} \leftarrow \emptyset$
4: **for** $i \leftarrow 0$ **to** $len(x) - 1$ **do**
5: $y \leftarrow x[0, i]$
6: $\mathcal{F} \leftarrow \{\psi \in \mathcal{P} \setminus \mathcal{S} \mid \textit{eb-mon}(y, \psi)$ `returns` $\top\}$
7: **if** $\mathcal{F} \neq \emptyset$ **then**
8: updatePoolInformation$(\mathcal{P} \setminus \mathcal{S}, \mathcal{F}, y)$
9: **if** l **then**
10: $has\_triggered \leftarrow \top$
11: $\mathcal{Y} \leftarrow$ generateTrainData(y)
12: $\phi \leftarrow$ extractDiscrFormula$(\mathcal{Y})$
13: **if** $\phi \neq$ `null` **then**
14: $far_\phi \leftarrow 0$
15: $\mathcal{P} \leftarrow \mathcal{P} \cup \{\phi\}$
16: **end if**
17: **break**
18: **else**
19: $\mathcal{S} \leftarrow \mathcal{S} \cup \mathcal{F}$
20: **end if**
21: **end if**
22: **end for**
23: **if** l **and not** $has\_triggered$ **then**
24: $\mathcal{Y} \leftarrow$ generateTrainData(x)
25: $\phi \leftarrow$ extractDiscrFormula$(\mathcal{Y})$
26: **if** $\phi \neq$ `null` **then**
27: $far_\phi \leftarrow 0$
28: $\mathcal{P} \leftarrow \mathcal{P} \cup \{\phi\}$
29: **end if**
30: **end if**
31: **end for**
32: **return** $\mathcal{P}$

Algorithm 3: UPDATEPOOLINFORMATION

input: pool $\mathcal{P}$ of formulas, set $\mathcal{F}$ of failure formulas, trace x

1: **for** $\phi \in \mathcal{F}$ **do**
2: $far_\phi \leftarrow (1-\alpha)\cdot$ NEWFAR$(\phi, x) + \alpha \cdot far_\phi$
3: **if** $far_\phi > far_{thr}$ **then**
4: REMOVE$(\phi, \mathcal{P})$
5: **end if**
6: **end for**
7: HANDLEREDUNDANCY$(\mathcal{P})$

During both phases, $\mathcal{P}$ is iteratively refined by (i) adding new formulas which are able to predict bad behaviors earlier and with increased reliability and coverage, and (ii) removing formulas that are ill-behaved or redundant. In addition to this refinement process autonomously operated by the framework, at any time, domain experts can, in principle, make changes to the pool $\mathcal{P}$, e.g., by manually specifying a new formula encoding a bad behavior.[3]

5.4.1 Warmup execution phase

In the *warmup* phase, we assume that a supervised learning training dataset $\mathcal{X}$ is available, consisting of pairs (x, l), where x represents a system execution trace of length $len(x)$, and l is its corresponding label ($\top$, if x is a trace ending with a failure; $\bot$ otherwise). The overall idea is to monitor, one after the other, all available system traces and, for each of them, to simulate its point-by-point arrival.

The *warmup* phase is dealt with by Algorithm 2. The procedure gets, as input, a pool $\mathcal{P}$ of bSTL formulas and the set $\mathcal{X}$ of training system traces. $\mathcal{P}$ may possibly be empty. This is the case when no formula is inserted into it by domain experts. For each training trace x, with label l, two variables are set: *has_triggered*, which keeps track of whether the framework has correctly identified the trace x as a failure one (when $l = \top$); and a set $\mathcal{S}$ of *suspended* formulas. $\mathcal{S}$ includes all formulas that, at some point, erroneously signalled a bad behavior in x (when $l = \bot$), and are thus ignored by the framework for its operation on the remainder of trace x.

Next, the framework starts the iterative part of its execution, during which the trace x is monitored sequentially, point-by-point. At each iteration i, with $0 \leq i \leq len(x) - 1$, the system restricts its attention to the prefix $y = x[0, i]$ of trace x, and it computes the set $\mathcal{F}$ of formulas leading to a *violation* (Algorithm 2,

[3]It is worth noticing that this manual specification can complement the *warmup* phase when limited supervised training data regarding the system are available.

line 6). To this end, it executes the monitoring algorithm `rtamt` that verifies each (non-suspended) formula in $\mathcal{P} \setminus \mathcal{S}$ against the current trace y. Since all formulas are meant to encode bad behaviors, we say that a formula ψ leads to a violation if *eb-mon*(y, ψ) returns $\top$ (*eb-mon* is defined in Section 5.2).

If at least one violation is detected, procedure updatePoolInformation is executed (Algorithm 3) to detect and remove redundant or non-reliable formulas from the pool, the latter being formulas issuing several false positives. The procedure will be described in detail later.

Then, if x is an actual failure trace, $\mathcal{P}$ is updated (Algorithm 2, lines 10–17) as follows. Training data to be used for the extraction of a new formula are generated by the function generateTrainData (Algorithm 2, line 11). The latter perturbs the execution trace y by adding random Gaussian noise as a counter-overfitting measure, thus producing a set of augmented traces $\mathcal{Y}$ of size n_{aug} (global parameter of the system). Next, function extractDiscrFormula (Algorithm 2, line 12) extracts a (bSTL) formula ϕ that discriminates between *normal* and *failure* (sub)traces obtained from those in $\mathcal{Y}$, by exploiting the evolutionary algorithm as described in Section 5.3. Notice that ϕ may be `null`, an event that, according to the proposed definition of EA, occurs if none of the formulas in the final front has an accuracy greater than 0.5. If ϕ is not `null`, the initial false alarm rate (FAR) of ϕ is set to 0, and the formula is added to $\mathcal{P}$ (Algorithm 2, lines 13–16).[4] At this point, since the trace x was recognized as a failure one by the framework, the execution on x is halted (Algorithm 2, line 17), and the framework is applied to the next trace in $\mathcal{X}$.

On the contrary, if the framework detected a violation and trace x was not a failure one, all involved formulas $f \in \mathcal{F}$ are *suspended*, meaning that they are not going to be considered by the framework for its execution on the remainder of trace x (Algorithm 2, line 19). This prevents them from repeatedly triggering the extraction of other ill-behaved formulas. Note how formulas in $\mathcal{F}$ are not immediately removed from $\mathcal{P}$, as such an approach would be too aggressive: their false positive detection might not be a generalized behavior, but something caused by random characteristics of trace x itself. As we will see, false positive detections are still considered by the procedure updatePoolInformation for the maintenance of the pool $\mathcal{P}$. Suspended formulas are reactivated when the next training trace is taken into account by the framework.

The iterative phase of the framework on trace x ends when either x is correctly recognized as a failure trace by a formula in $\mathcal{P}$, or x has run out of points without

[4]The False Alarm Rate (FAR) is expressed as the ratio between the number of negative events wrongly categorized as positive (false positives) and the total number of actual negative events (false positives + true negatives). Recall that, in our setting, a positive event represents a failure of the monitored system. Formulas with a low FAR are to be preferred.

any failure detection. In the latter case, if trace x was a failure one, we force the formula extraction process (Algorithm 2, lines 23–30). As the last operation of the framework (Algorithm 2, line 32), after being run on every training system trace, the obtained monitoring pool $\mathcal{P}$ is returned.

We would like to conclude this account of the operation of Algorithm 2 by observing that the *warmup* mode draws inspiration from the *teacher forcing* technique employed in deep learning [184]. Such an approach is used here to correct both false positive (Algorithm 2, line 9) and false negative (Algorithm 2, line 19 and line 23) framework predictions. For instance, as we already pointed out, the framework starts its execution with a possibly empty pool $\mathcal{P}$ of properties. Thus, in the most extreme case ($\mathcal{P} = \emptyset$), it cannot identify any bad behaviors of the system. In such a case, the failure is "detected" by observing the training label associated with the training execution trace, an event that forcedly triggers the pool update process. Intuitively, the whole scenario can be thought of as having an oracle assisting and instructing the framework. It is to be expected that, over time, the pool $\mathcal{P}$ becomes large enough so as to allow for the effective detection of bad behaviors of the system, progressively substituting the oracle in its role of correcting false positive and false negative predictions.

Let us focus now on the procedure updatePoolInformation$(\mathcal{P}, \mathcal{F}, x)$ (Algorithm 3). Operationally, for each formula $\phi \in \mathcal{F}$ that leads to a violation, the corresponding FAR $far_\phi \in [0, 1]$ is updated. Formulas whose FAR crosses a given threshold far_{thr} (a global parameter of the framework) are removed from the monitoring pool (Algorithm 3, lines 3–4). As already pointed out, a FAR equal to 0 is associated with every formula when added to $\mathcal{P}$. Then, the value of FAR is suitably updated according to the exponential moving average with smoothing constant α, which takes into account the "historical" FAR and the new FAR of the formula (Algorithm 3, line 2); the latter is considered to be 0 if the triggered formula actually anticipated a failure, 1 otherwise (false positive case). In the absence of detailed historical data, assigning an initial FAR equal to 0 to the formulas in the pool is a sensible choice, as they are either defined by a domain expert or generated by the evolutionary algorithm, which in turn optimizes accuracy and robustness measures.

The choice of relying on FAR for the pool maintenance instead of on other "symmetric" performance metrics, say F1-score, is twofold. First, formulas providing several false detections may cause a degradation of the monitoring pool, where other ill-founded formulas are added as a result of their triggering. Thus, they should be avoided at all costs. On the contrary, formulas leading to false negatives do not bring any adverse effect on the monitoring pool, except for increasing its size. The second reason pertains to the very nature of F1-score and similar metrics. To calculate it,

it is necessary to establish when a formula experiences both false positives and false negatives. False positives, that is, false detections of bad behaviors, can be easily recognized: if a formula triggers and the forecasted bad event does not occur, that can be unequivocally considered as a false positive. The detection of false negatives is, instead, more subtle, and not well-defined. Indeed, it is perfectly admissible for the system to encounter a total failure not anticipated by any formula in the pool, since they may correctly model completely different failure scenarios. In that case, formulas should not be penalized for the missed detection.

Finally, procedure HANDLEREDUNDANCY($\mathcal{P}$) (Algorithm 3, line 7) removes redundant formulas, i.e., it detects groups of formulas with similar behavior and keeps a single representative for the entire group (the formula with the lowest FAR or the newest one, if the FAR is the same). To detect the similarity of two formulas, we rely on the Jaccard/Tanimoto test [45] that compares the histories of failures flagged by the formulas along the framework execution.

As a last remark, note how procedure UPDATEPOOLINFORMATION, in the way it is used by Algorithm 2, allows us to continuously update the monitoring pool $\mathcal{P}$ as the training instances are processed, ensuring its quality.

Algorithm 4: Framework execution (*runtime* phase)

input: initial non-empty pool $\mathcal{P}$ of formulas, non-empty set $\mathcal{G}$ of good behavior training traces, incoming system trace x

```
1:  while true do
2:    F ← {ψ ∈ P | eb-mon(x, ψ) returns ⊤}
3:    if F ≠ ∅ then
4:      handleRedundancy(P)
5:      X ← generateTrainData(x)
6:      φ ← extractDiscrFormula(X)
7:      if φ ≠ null then
8:        far_φ ← FAR(G, φ)
9:        if far_φ ≤ far_thr then
10:         P ← P ∪ {φ}
11:       end if
12:     end if
13:   end if
14: end while
```

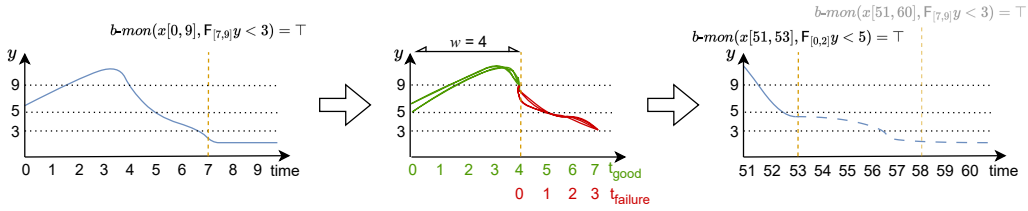


Figure 5.2: A simplified example of the framework operation. Good and failure behavior subtraces are represented in green and red, respectively.

5.4.2 Runtime execution phase

Let us now concentrate on the *runtime* phase of the framework which is implemented by Algorithm 4. Here, the framework is used to continuously monitor an incoming trace x, generated by a system during its execution. Other than the trace, the procedure gets in input a pool $\mathcal{P}$ of properties, that can be assumed to be non-empty, either because it is returned by Algorithm 2 or hand-filled by domain experts. In addition, it takes into consideration a non-empty set $\mathcal{G}$ of past good execution traces of the system. The latter can be either extracted from the training warmup data, if available, or derived directly from the execution history of the system, restricting to those portions that are sufficiently far from failure events, following the suggestions of domain experts.

Algorithm 4 behaves as follows. At each time step, the set $\mathcal{F}$ of formulas leading to a *violation* is computed (Algorithm 4, line 2) by executing the monitoring algorithm `rtamt`, which checks each formula in $\mathcal{P}$ against the incoming system trace x. If at least one formula is triggered, $\mathcal{P}$ is updated (Algorithm 4, lines 3–13). First, procedure HANDLEREDUNDANCY is called, to identify and remove from the pool $\mathcal{P}$ possible redundant formulas, exactly in the same way as in the warmup phase. Next, training data to be used for the extraction of a new formula are generated by the function GENERATETRAINDATA(x). Finally, function EXTRACTDISCRFORMULA($\mathcal{X}$) extracts a bSTL formula ϕ that discriminates between *normal* and *failure* (sub)traces by using the EA. If the formula ϕ generated by the EA is not `null`, its FAR is computed with respect to the reference set of good traces $\mathcal{G}$ and, if such a value is less than or equal to the threshold far_{thr}, the formula is added to the pool $\mathcal{P}$.

As it can be noticed, the main differences between the warmup and runtime phases are that, in the latter, there is no teacher forcing, and thus the entire failure detection task is carried out by means of monitoring; moreover, the FAR of a formula is established only once by considering all traces in the reference set $\mathcal{G}$, being the latter fixed.

As a final remark, note that Algorithm 4 (runtime) can, in principle, be run independently from Algorithm 2 (warmup), if there is at least one property in $\mathcal{P}$ and a set of good traces $\mathcal{G}$ of the considered system is available. This allows one to use the framework in a runtime setting even in the absence of supervised training data, as long as at least one failure property has been provided by domain experts and some (portions of) unlabeled past execution traces of the system, that express good/normal behaviors, are accessible.

An intuitive account of the operation of the framework is depicted in Figure 5.2. The framework is first attached to a trace x generated by the system for its runtime monitoring, with a pool $\mathcal{P}$ containing just the formula $\phi = \mathsf{F}_{[7,9]} y < 3$ (left picture). Function *eb-mon*(x, ϕ) is run against the incoming trace and, specifically, *b-mon*$(x[0, 9], \phi)$ identifies a failure occurring at time point 7. This leads to the extraction of a new formula. To this end, trace $x[0, 7]$ is augmented, and then the EA is run on the set of resulting traces. In the middle picture, just for illustrative purposes, an exemplary splitting of the augmented traces based on a window length $w = 4$ is reported. Each trace is partitioned into a good behavior prefix and a failure suffix. For formula evaluation purposes, in the EA each subtrace is considered as to be starting from index 0. As a result, the formula $\psi = \mathsf{F}_{[0,2]} y < 5$, which is able to distinguish between the augmented prefixes and suffixes, is generated and added to the pool $\mathcal{P}$. Finally, a subsequent part of the operation of the framework is described (right picture) Here, the recently discovered formula ψ identifies a failure occurring at time point 53 (*b-mon*$(x[51, 53], \psi) = \top$). Without such a formula, ϕ would have detected a violation only with respect to time point 58.

For the sake of convenience, all the global parameters of the framework are listed in Table 5.1, with an intuitive account and a short description of their expected behavior.

5.5 Experimental evaluation

In this section, we give a detailed account of the experimental evaluation of the framework on 3 public datasets. In addition, we make a comparison with previous results from the literature. First, we introduce the datasets; then, we describe the experimental workflow; finally, the obtained results are portrayed. We pay particular attention to interpretability issues.

Table 5.1: Global parameters of the monitoring framework.

	Description	Value	Search range	Expected behavior
$\boldsymbol{\alpha}$	*smoothing constant* for formulas FAR update	0.9	$\{0.7, 0.8, 0.9\}$	A lower value gives greater weight to new scores. Such a behavior is preferred, for instance, in the event of a system undergoing rapid changes.
$\boldsymbol{far_{thr}}$	*maximum allowed FAR* for formulas in the pool	0.2	$\{0.1, 0.2, 0.3\}$	A low value leads to more reliable formulas being kept in the monitoring pool. Still, in the presence of noise/outliers in the monitored traces, it may cause a formula to be inadvertently removed from the pool. There, a higher value should be considered.
$\boldsymbol{n_{aug}}$	*number of augmentations* for each failure trace	100	$\{50, 100, 150\}$	A large value should help avoid overfitting; however, it also increases the computational burden of the formula extraction phase.

5.5.1 Datasets

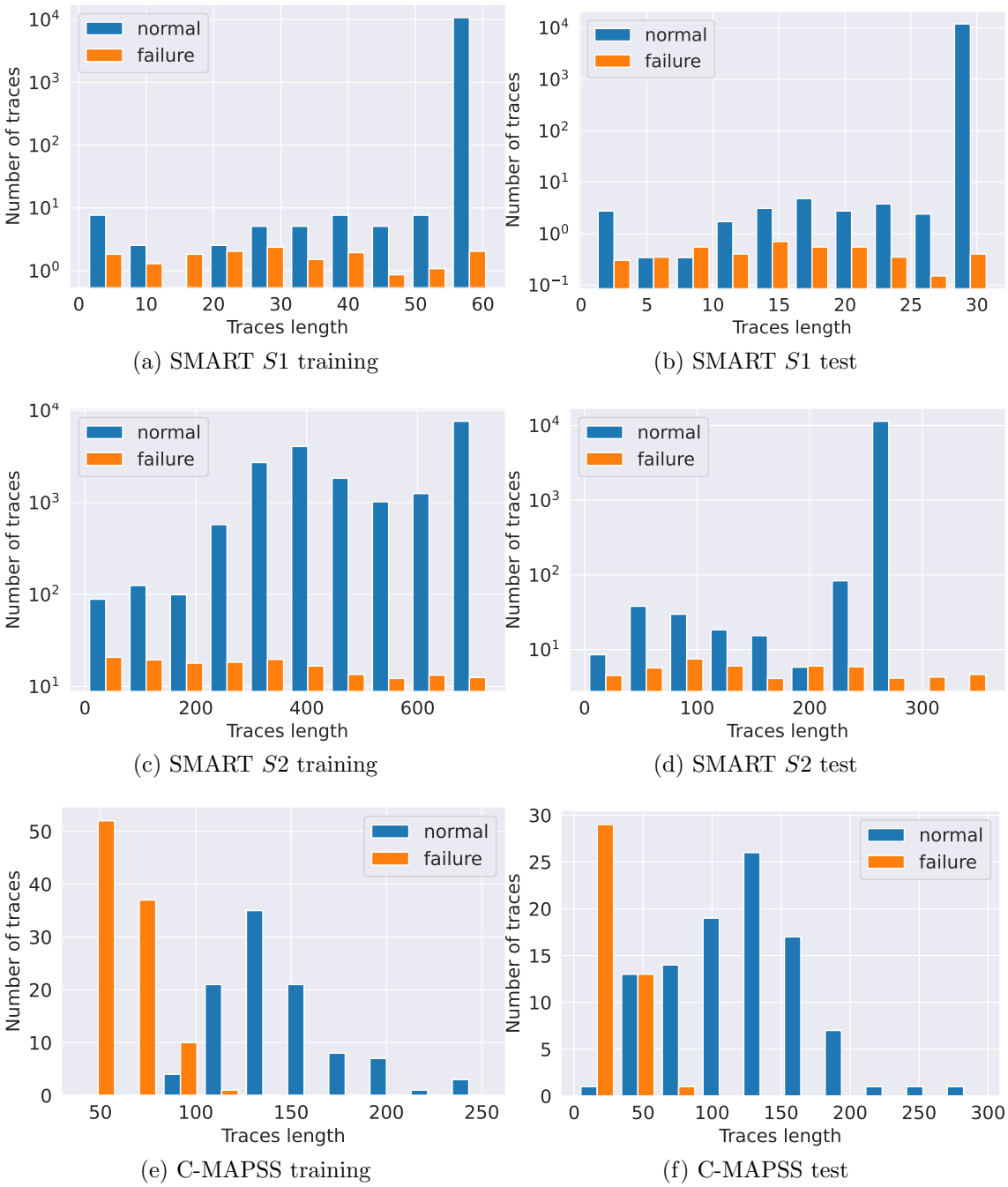
We considered the datasets Backblaze Hard Drive[5], Tennessee Eastman Process[6], and NASA C-MAPSS[7].

The *Backblaze Hard Drive* dataset (also referred to as *SMART* dataset hereafter) contains continuously updated information on the "health" status of hard drives in the Backblaze data center. Here, we focus on Self Monitoring Analysis and Reporting Technology (SMART) attributes of the ST4000DM000 hard drive model recorded daily from 2015 to 2017. Each trace is described by the following features: the date of the report, the serial number of the drive, a label indicating a drive failure and 21 SMART parameters with both discrete and real values. To compare

[5] https://www.backblaze.com/b2/hard-drive-test-data.html

[6] https://doi.org/10.7910/DVN/6C3JR1

[7] https://data.nasa.gov/dataset/C-MAPSS-Aircraft-Engine-Simulator-Data/xaut-bemq



(a) SMART $S1$ training

(b) SMART $S1$ test

(c) SMART $S2$ training

(d) SMART $S2$ test

(e) C-MAPSS training

(f) C-MAPSS test

Figure 5.3: Trace length distributions.

the framework with the literature, two training/test set splits are considered:

- Split $S1$: training set from October to November 2016 with 0.68% failure traces, test set December 2016 with 0.64% failure traces. The total number of traces is 34970, and their length distributions are depicted in Figure 5.3;
- Split $S2$: training set from January 2015 to December 2016 with 0.71% failure traces, test set January to December 2017 with 0.66% failure traces. The total number of traces is 36242, and their length distributions are depicted in Figure 5.3;

The *Tennessee Eastman Process* (TEP) dataset contains simulated data from a fictitious chemical plant. This dataset includes 1000 training and 1000 test traces labelled with Type 0 (normal behavior) or Type 1 (faulty behavior) sampled every 3 minutes. Each training trace lasts for 25 hours, whereas test traces last for 48 hours. There are 500 faulty traces in both sets. Each simulation is represented by a multivariate time series with the following features: trace ID, fault type, and 52 variables tracking data about the operating values of plant components.

The *NASA Commercial Modular Aero-Propulsion System Simulation* (C-MAPSS) dataset includes run-to-failure simulated data of turbofan jet engines. Specifically, in the considered dataset FD001, engines are simulated according to a single operating condition (called *Sea level*) and their failures are attributable to one possible cause (HPC degradation). Each engine simulation is represented by a multivariate time series obtained from 21 engine sensors. Although each trace represents the simulation of a different engine, the data can be considered to be from a fleet of engines of the same type. Data are sampled at one value per second, and the trace length distributions are depicted in Figure 5.3. The dataset includes 100 training traces, each ending with a failure, and 100 test traces, each ending an arbitrary and known number of time steps before the failure (gap). In order to compare our framework with the literature [93], failure traces are generated by considering the 30% suffix of each engine observation as faulty, and the remaining 70% prefix as normal behavior. Thus, this leads to 200 training traces. On the other hand, 43 failure and 100 normal behavior traces are generated for the test set. The reason is that, given a test set trace, the 30% suffix is computed over the trace length including the gap and, thus, it may result to be empty.

5.5.2 Experiment setup

For each dataset, we performed the initial warmup phase by running Algorithm 2 on a sample of training execution traces related to both malfunctions (failure traces)

and good executions. The traces were considered by the framework one after the other, according to a random ordering.

Once the warmup phase ended, the framework was evaluated on test set traces (Algorithm 4) in two modes: the *online* mode, where the framework continues to learn new properties from the execution traces, and the *offline* mode, where the properties in the monitoring pool are not updated, so that only the properties learnt in the warmup phase are taken into account when predicting failures on test set traces. This latter mode was useful to compare the proposed solution with those from the literature, while the former let us determine how the values of the considered metrics evolved over time. The two test set evaluation modes were carried out on a random ordering of both good and failure traces.

The performance of the two test phases was evaluated in terms of *precision* (P), *recall* (R), *FAR*, and *F1-score* ($F1$). Let T_P be the number of true positives, that is, bad behaviors identified as such, F_P be the number of false positives, T_N be the number of true negatives, and F_N be the number of false negatives. The metrics are defined as follows:

$$P = \frac{T_P}{T_P + F_P} ,$$

$$R = \frac{T_P}{T_P + F_N} ,$$

$$FAR = \frac{F_P}{F_P + T_N} ,$$

$$F1 = 2 \cdot \frac{P \cdot R}{P + R} .$$

All experiments were run 10 times varying the random seed governing the order in which execution traces are presented to the framework during the warmup and the online test phases, so as to collect statistical data regarding the considered metrics.

5.5.3 Results

To begin with, the global parameters of the framework, chosen through grid search optimization on training set data, are shown in Table 5.1. In the remainder of the section, we will assess the framework performance in several respects. First, we will present the results of the *offline* and the *online* evaluation, as described in Section 5.5.2. Then, we will focus on the impact of teacher forcing and formula *max horizon*.

Table 5.2: Experimental results of the monitoring framework.

Dataset	Approach	Precision	Recall	FAR	F1-score
SMART S1	[110]	0.87	0.41	0.00	0.55
	[82]	0.51	0.54	0.00	0.52
	our	0.54	0.60	0.00	0.56
SMART S2	[110]	0.98	0.87	0.01	0.92
	[173]	0.91	0.94	0.05	0.93
	our	0.89	0.97	0.00	0.93
TEP	[74]	1.00	1.00	–	1.00
	[139]	1.00	1.00	0.00	1.00
	our	1.00	1.00	0.00	1.00
C-MAPSS	[93]	0.71	1.00	–	0.83
	our	0.96	0.77	0.01	0.86

Note: the results of [82], [173], [74], [139], and [93] are listed as reported in the original references; those of [110] have been determined by the authors of this work running the code made available in the original publication on the considered datasets. Numbers are rounded to two decimal places.

Offline evaluation

As for the *offline evaluation* mode, we compared the proposed solution with other state-of-the-art approaches to failure detection on the three previously-described datasets. Given the continuously updating nature of the Backblaze dataset, we focused our analysis on two studies that take into account the specific versions we consider, namely, those reported in [82] and [173]. The first one [82] makes use of a feed-forward neural network model on split $S1$, while the second one [173] evaluates a Long-Short Term Memory (LSTM) recurrent neural network on split $S2$. In addition, we took into account a third proposal [110], that is, a model obtained by combining a convolutional neural network (CNN) and an LSTM recurrent neural network, applying it to both $S1$ and $S2$ splits, following the setup outlined by the authors for the SMART features group. As for the case of fault detection on the TEP dataset, we considered an approach based on image processing techniques along with feed-forward and radial basis function neural networks [74], and a solution based on a nonlinear support vector machine [139]. Finally, as for the C-MAPSS case study, we compared our framework with a solution based on a CNN model presented in [93].

Results achieved by the above solutions are reported in Table 5.2, together with

those of the proposed framework (label *our*). In terms of F1-score, our solution exhibits an average performance on par with the considered state-of-the-art ones. This is even more relevant if we also bear in mind that all the contenders provide no explanation for the predicted failures. On the contrary, a distinguishing feature of the proposed approach, compared to previous ones, is that it is interpretable: it relies on the extraction of properties expressed as temporal logic formulas, that provide an understandable explanation of the undesired behaviors of the system. Moreover, they can be subsequently exploited for tasks such as root cause analysis, diagnosis, and preemptive failure detection.

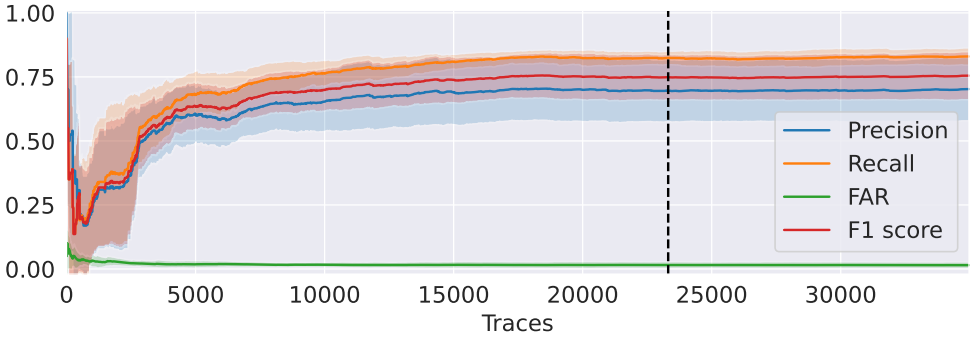
Online evaluation

As for the *online evaluation* mode, results for the SMART, TEP, and C-MAPSS datasets are shown in Figure 5.4. Note that, as the number of traces seen by the framework increases, a slight but consistent improvement of the metrics occurs. This is not obvious, since in such a case maintaining a good performance requires the ability to discover new properties able to reflect the evolution of the behavior of the monitored system over time.

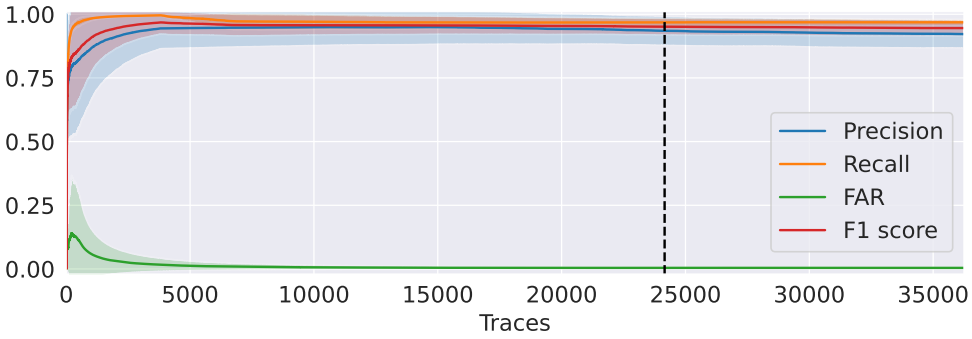
Figure 5.5 illustrates, for each considered dataset, the average number of formulas in the monitoring pool at each warmup and runtime iteration. Note that, at certain iterations, there is a decrease in the pool size. This happens when formulas are removed because they are redundant.

As an example of removal from the pool, we present the case in which two properties, $\mathsf{F}_{[30,45]} SENSOR_{11} < 49.60$ and $\mathsf{F}_{[36,41]} SENSOR_{11} < 49.77 \wedge \mathsf{F}_{[45,52]} SENSOR_{11} < 49.39$, were extracted in a framework execution on the C-MAPSS dataset. After a series of iterations, their failure detection histories were, respectively, $[1,0,0,1,0,1,1,0,0,1,0,1,1,0,0,1,1,1]$ and $[1,0,1,0,0,1,1,0,1,0,1,1,1]$, while the FAR values were 0.07 and 0.0. Since both properties were considered to be redundant according to the Jaccard/Tanimoto test, the framework kept only the second one, by reason of its lower FAR. From a domain perspective, the two formulas refer to a similar behavior of the same sensor ($SENSOR_{11}$) which indicates a loss of static pressure in the *high-pressure compressor outlet*.

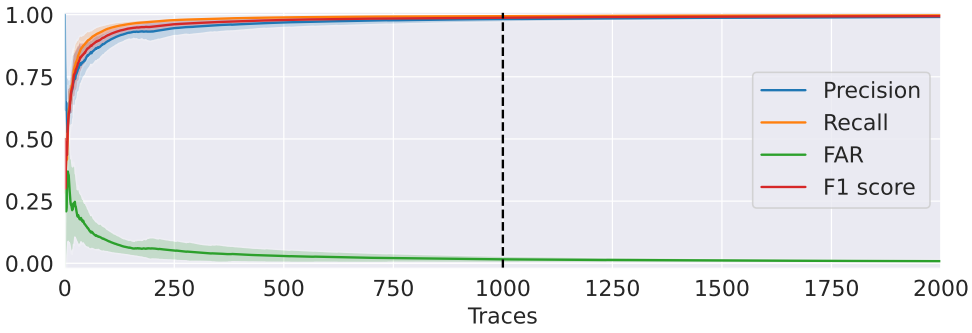
Let us now consider some examples of the bSTL formulas used within the framework. An example for the Backblaze dataset is formula $(\mathsf{G}_{[0,2]} SMART_{194} > 45.6) \wedge (\mathsf{F}_{[2,3]} SMART_{198} > 0.32)$. Such a formula makes evident a bad behavior where the hard drive maintains a temperature exceeding 45.6 °C in the first 3 days, and then, in the following 2 days, its *uncorrectable sector count* becomes greater than 0. As another example of framework execution, consider the formula $f_1 = \mathsf{F}_{[0,19]} SMART_{198} > 2.59$, extracted (and added to the monitoring pool) during an iteration of the frame-



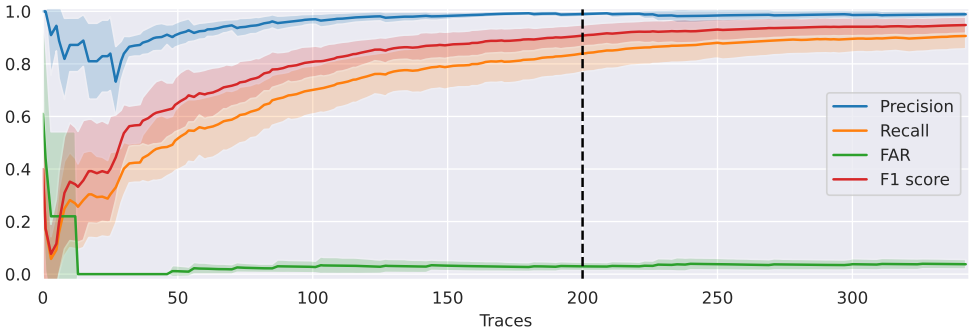
(a) SMART $S1$



(b) SMART $S2$



(c) TEP



(d) C-MAPSS

Figure 5.4: Metrics average and standard deviation for the considered datasets. The vertical dashed line represents the transition from warmup to online traces.

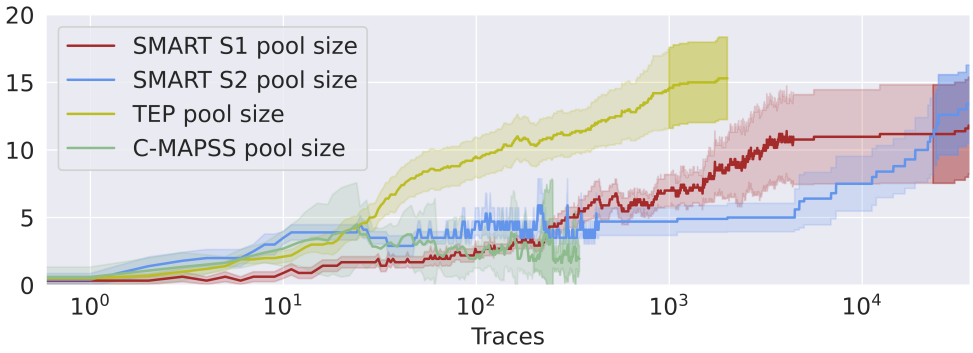


Figure 5.5: Average and standard deviation of the pool size for each framework iteration in the case of the SMART S1, SMART S2, TEP, and C-MAPSS datasets, in both warmup (transparent area) and runtime (opaque area) phases. The x-axis is on a logarithmic scale.

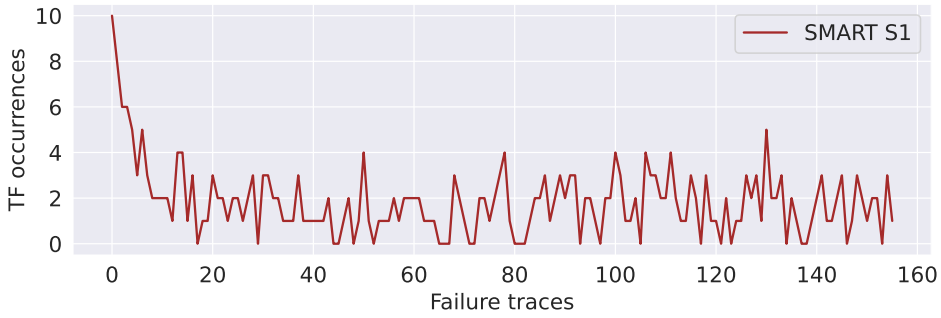
work. According to the definitions of the SMART attributes, sensor $SMART_{198}$ is a critical one and f_1 expresses the fact that the threshold 2.59 of *sector read/write errors* is exceeded. During a later iteration of the *warmup* phase, a failure prediction is issued thanks to the triggering of f_1. As a consequence, $f_2 = \mathsf{F}_{[1,16]} SMART_{189} > 8.28$ is extracted, meaning that a certain number (8.28) of *unsafe fly height conditions* is reached before the critical number of sector read/write errors is exceeded. This pattern is quite reasonable, as it describes a case in which the disk head is operating at an unsafe height, ultimately damaging a disk sector and consequently causing read and write errors. Notice that the framework allows us to predict a failure based on sensor $SMART_{189}$, which is not considered to be critical in the SMART specification, by uncovering a pattern linking it to the critical sensor $SMART_{198}$.

Turning to the TEP dataset, an extracted formula is $(\mathsf{G}_{[1,4]} XMEAS_{21} > 94.6) \wedge (\mathsf{G}_{[2,4]} XMEAS_{20} > 341)$. It reveals a bad behavior where the *compressor* is operating with a power greater than 341 kW, while the temperature of the *reactor of the plant* exceeds 94.6 °C.

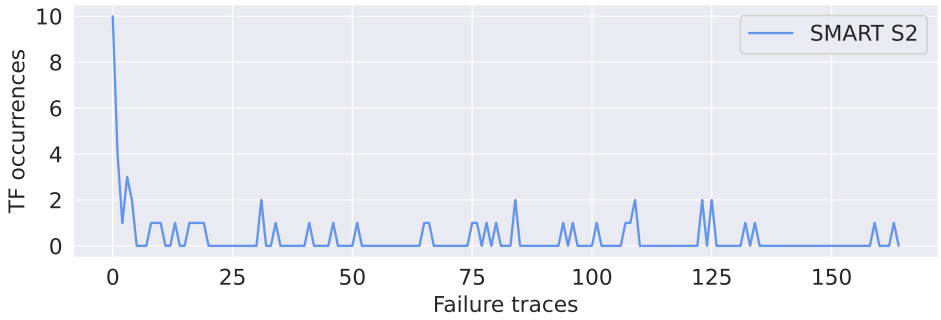
As for the C-MAPSS dataset, the formula $(SENSOR_{10} < 1.3) \wedge (\mathsf{F}_{[4,6]} SENSOR_{11} < 47.62)$ was generated, which signals a bad behavior where a loss of pressure in the *high-pressure compressor outlet* follows a loss of pressure in the *engine*. Notice that the subformula $SENSOR_{10} < 1.3$ does not contain any time operator, meaning that it is evaluated at the currently observed time point.

Impact of teacher forcing

Figure 5.6 reports the number of teacher forcing interventions during the warmup phase, as more and more false positives and negatives are encountered. Specifically,



(a) SMART $S1$



(b) SMART $S2$

(c) TEP

(d) C-MAPSS

Figure 5.6: Teacher forcing interventions during the warmup phase. For each amount of encountered failure traces, the sum over multiple (10) framework executions is reported.

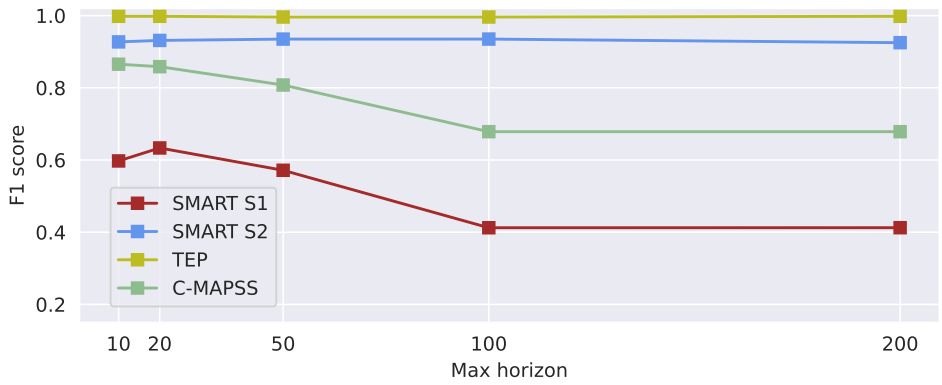


Figure 5.7: Impact of maximum horizon value in test F1-score for different datasets.

for each amount of encountered failure traces, the sum over multiple (10) framework executions is reported. As expected, teacher forcing triggers mainly at the beginning of the warmup phase, when the monitoring pool is empty. As formulas are learned over time, teacher forcing interventions decrease till a stationary behavior is reached. Of course, the latter depends on the specific dataset, and it confirms what was to be expected from the performance reported in Table 5.2. As an example, on the TEP dataset, where an F1-score of 1.0 is achieved, the number of teacher forcing interventions rapidly approaches 0.

Impact of max horizon

Figure 5.7 reports, for a single execution of the framework, the offline mode performance on each dataset, as obtained by varying the *max horizon* value. Once more, different datasets exhibit different behaviors. Although results might appear rather counterintuitive (setting a large *max horizon* does not prevent, in principle, the discovery of formulas with shorter horizons), following a preliminary analysis, they are likely due to an overfitting effect. Indeed, formulas with a larger horizon have the capability of capturing more detailed and extended phenomena, that could be highly trace-dependent. Moreover, we would like to recall that the concept of horizon does not apply to the framework in general, but is tied to the particular kind of logic and monitoring tool employed here.

5.6 Discussion

The proposed framework relies on approaches originating from the two fields of machine learning and formal methods, combining their strengths in an effective way. More precisely, the former domain provided us with tools and techniques for the

extraction of properties from temporal data, while the latter allowed us to formalize such properties by means of logic formulas and to online monitor a given system against them in a principled manner. The key feature of the proposed approach is its interpretability: as shown in Section 5.5.3, by means of the extracted logical formulas, the framework gives an understandable account of settings leading to future failures, allowing domain experts to take appropriate action and enriching their overall knowledge. While contributions from the literature show that interpretability is often achieved at the expense of prediction accuracy, e.g., by relying on a simple white-box model instead of a more complex black-box one, quantitative results showed that the performance of the proposed approach is on par with previous, non-interpretable solutions.

As a final note, while in this work we applied the proposed framework to the domain of failure detection, similar ideas can in principle be employed to detect and predict any type of event or anomaly, whether positive or negative in nature. Among the first, we mention a spike in the history of sales of a retail store, or a generalized increase in the grade point average of students enrolled in the latest edition of a course; among the latter scenarios, the detection of seizures in hospitalized patients based on their continuously recorded vital signs, or the identification of violations of a level of service agreement in the context of a contract between a service provider and a customer.

We would like to conclude this section by pointing out some limitations of the framework. First, in the considered datasets, all traces come from the same plant (resp. hard disk, jet engine model) operating under the same conditions. To deal with more than one type of system, separated monitoring pools have to be employed to prevent conflicts among formulas. Second, the considered datasets only deal with numerical data. It is worth evaluating the proposed approach on datasets encompassing categorical data, naturally leading to the usage of other logics like, for instance, LTL. Third, Algorithm 4 operates in a sequential fashion: (i) the system is monitored until a formula is satisfied by the incoming data; (ii) such an event triggers the phase of the property extraction, that results in the addition of a new formula to the pool; (iii) then, the monitoring of the system resumes. Although this behavior is perfectly acceptable for a prototype implementation applied on benchmark datasets, as the one described here, a multithreaded version, able to update the property pool asynchronously, while monitoring the system, remains to be developed. Finally, Algorithm 4 makes use of a fixed reference set of good behavior traces to prevent formulas with a high FAR to be added to the monitoring pool. This is definitely a reasonable approach, but it does not take into account changes in the behavior of the monitored system, that may happen due to, for instance, updates, upgrades, or degradation phenomena. To overcome this limitation, we may think of extracting

new normal behavior traces from runtime data and adding them to the reference set.

Conclusions

In this thesis, we addressed three main problems related to the employment of sensor systems, namely sensor selection, virtual sensing of temperature in indoor environments, and the design of a monitoring and machine learning framework for early failure detection.

In the first part, we analyzed the problem of sensor selection and placement in virtual sensing of temperature. We proposed and evaluated different methodologies and solutions based on evolutionary algorithms and machine learning techniques. Our results showed that the proposed models are able to achieve satisfactory performance using a limited set of optimally placed predictor sensors in the environment. These attained findings have been published in [28] that appeared in the open access journal MDPI Sensors.

The second part of our work was dedicated to the development and evaluation of different virtual sensing strategies for temperature prediction. We conducted a comprehensive assessment of these strategies, including novel solutions based on recurrent neural networks and graph neural networks, able to effectively exploit spatio-temporal features. Our results showed that our proposed models are able to accurately complete the information coming from real physical sensors, allowing us to effectively carry out monitoring tasks such as anomaly or event detection. This research activity led to the drafting and presentation of [26] to the 13th International Workshop on Spatial and Spatiotemporal Data Mining (abbreviated SSTDM, in cooperation with IEEE ICDM 2020). In this work, we analyze the performance of various techniques, including simple average and inverse distance weighted average, particle filters, linear regression, extreme gradient boosting regression, and long short-term memory recurrent neural networks.

In the final part of our work, we proposed a monitoring and machine learning framework for early failure detection. This framework extracts interpretable properties, expressed in a given temporal logic formalism (STL) from sensor data, and leveraged monitoring techniques to detect critical behaviors of a system that could lead to a failure. Our proposed framework was evaluated through an experimental assessment performed on benchmark datasets, and then compared to previous approaches from the literature. In terms of contributions, the developed framework was presented at the Workshop on Reasoning about ACtions and Events over Streams (RACES), that has been held as part of KR 2020. Then, an experiment carried out on the NASA C-MAPSS dataset containing run-to-failure execution traces of

turbojet engines led to the submission of [25] to the 2nd Workshop on Artificial Intelligence and formal VERification, Logic, Automata, and sYnthesis (OVERLAY) @ BOSK 2020. After a series of analysis, design, development, and evaluation cycles, the final and extended version of the framework was thoroughly discussed in detail through its publication [24] in the open access journal IEEE Access.

Looking ahead, there are still several open research directions in these areas. For instance, for sensor selection and virtual sensing in indoor environments, we plan to develop other models, such as temporal graph networks, to improve predictive accuracy or perform tasks such as temperature trend simulation. In detail, this kind of models is able to exploit historical data to learn the dynamics of the environment, such as the rate of heat loss through walls or the amount of heat generated by a stove. This information can then be used to make short- and long-term predictions about the temperature in any part of the room. Providing physical simulations with first-principle approaches is an expensive activity in terms of computational resources and work required for the creation of numerical models related to different considered environmental configurations. Although the usage of transductive graph networks has proven effective in approximating such simulation data, as a future research direction it could be profitable to deeply investigate the generalization capacity of such models for monitoring environments with different configurations. In particular, focusing on an inductive learning approach able to provide a fair trade-off between accuracy and generalizability. This last factor is of paramount importance, considering that it would allow to reduce the amount of physical simulations needed to train the models.

Finally, for the monitoring and machine learning framework, we plan to extend our framework to other types of critical systems and evaluate its performance in real-world scenarios. We also plan to investigate the use of reinforcement learning techniques to improve the adaptability of our framework to changing system conditions. Furthermore, given its modular and extensible nature, we plan to extend our approach using other logical formalisms capable of expressing spatio-temporal properties in order to exploit the information learned from graph networks such as those studied in the case of virtual sensing of indoor environments. Another potential extension of this work involves conducting a more comprehensive investigation into the integration of extracted logical formulas, their structure, and their interaction with offline knowledge. This future research avenue aims to delve deeper into the intricate relationships between logical formulas derived from the framework and how they interface with existing knowledge. By examining the interplay between these logical structures and offline information, we can gain valuable insights into the broader applicability and generalizability of our findings, as well as potentially uncover novel connections that further enhance the understanding and utilization

of monitoring systems in different practical scenarios.

In conclusion, this thesis has contributed to the advancement of smart sensor technology, by proposing novel solutions and methodologies. In today's world, the use of sensors has become ubiquitous, with various industries adopting smart sensing technology to enhance their products and services. This technology involves the integration of advanced sensors and data analytics techniques, which allow for real-time monitoring, prediction, and diagnostic. The integration of the latter offers various benefits, including increased efficiency, cost savings, improved accuracy and safety. We hope that our work will inspire and guide further research in these areas, ultimately leading to more accurate, efficient, and effective sensor-based systems.

Bibliography

[1] L. Aceto, A. Achilleos, A. Francalanza, A. Ingólfsdóttir, and K. Lehtinen. Adventures in monitorability: from branching to linear time and back again. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

[2] L. Aceto, A. Achilleos, A. Francalanza, A. Ingólfsdóttir, and K. Lehtinen. An operational guide to monitorability. In *Proceedings of the 17th International Conference on Software Engineering and Formal Methods (SEFM 2019)*, volume 11724, pages 433–453. Springer, 2019.

[3] K. Aggarwal, O. Atan, A. K. Farahat, C. Zhang, K. Ristovski, and C. Gupta. Two birds with one network: Unifying failure event prediction and time-to-failure modeling. In *Proceedings of the 6th IEEE International Conference on Big Data (BigData 2018)*, pages 1308–1317. IEEE, 2018.

[4] E. Alpaydin. *Introduction to machine learning*. MIT press, 2020.

[5] M. Alshiekh, R. Bloem, R. Ehlers, B. Könighofer, K. Niekum, and U. Topcu. Safe reinforcement learning via shielding. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence, (AAAI 2018)*, pages 2669–2678. AAAI Press, 2018.

[6] J. D. Anderson and J. Wendt. *Computational fluid dynamics*. McGraw-Hill Education, 1995.

[7] N. Aussel, S. Jaulin, G. Gandon, Y. Petetin, E. Fazli, and S. Chabridon. Predictive models of hard drive failures based on operational data. In *Proceedings of the 16th IEEE International Conference on Machine Learning and Applications (ICMLA 2017)*, pages 619–625. IEEE, 2017.

[8] T. Bäck, D. B. Fogel, and Z. Michalewicz. *Evolutionary Computation 1: Basic Algorithms and Operators*. CRC press, 2018.

[9] Backblaze hard drive stats, Backblaze, Inc. `https://www.backblaze.com/b2/hard-drive-test-data.html`, 2023. Accessed: 2023-03-24.

[10] N. Baharisangari, J. R. Gaglione, D. Neider, U. Topcu, and Z. Xu. Uncertainty-aware Signal Temporal Logic inference. In *Proceedings of the 13th*

Working Conference on Verified Software: Theories, Tools, and Experiment (VSTTE 2021), volume 13124, pages 61——85. Springer, 2021.

[11] E. Bartocci, J. Deshmukh, A. Donzé, G. Fainekos, O. Maler, D. Ničković, and S. Sankaranarayanan. Specification-based monitoring of cyber-physical systems: A survey on theory, tools and applications. In *Lectures on Runtime Verification*, pages 135–175. Springer, 2018.

[12] E. Bartocci, Y. Falcone, A. Francalanza, and G. Reger. Introduction to runtime verification. In *Lectures on Runtime Verification*, pages 1–33. Springer, 2018.

[13] P. W. Battaglia, J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner, et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.

[14] BayesLTL GitHub reference page. `https://github.com/IBM/BayesLTL`. Accessed: 2023-03-24.

[15] J. Bergstra, D. Yamins, and D. Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *Proceedings of the 30th International Conference on Machine Learning (ICML 2013)*, volume 28, pages 115–123. PMLR, 2013.

[16] C. M. Bishop and N. M. Nasrabadi. *Pattern recognition and machine learning*, volume 4. Springer, 2006.

[17] G. Bombara and C. Belta. Offline and online learning of signal temporal logic formulae using decision trees. *ACM Transactions on Cyber-Physical Systems*, 5(3):1–23, 2021.

[18] G. Bombara, C. I. Vasile, F. Penedo, H. Yasuoka, and C. Belta. A decision tree approach to data classification using Signal Temporal Logic. In *Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control (HSCC 2016)*, pages 1–10. ACM, 2016.

[19] J. d. Borda. Mémoire sur les élections au scrutin. *Histoire de l'Academie Royale des Sciences pour 1781*, 1784.

[20] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[21] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and regression trees*. Routledge, 2017.

[22] D. Bresolin, E. Cominato, S. Gnani, E. Muñoz-Velasco, and G. Sciavicco. Extracting interval temporal logic rules: A first approach. In *Proceedings of the 25th International Symposium on Temporal Representation and Reasoning (TIME 2018)*, volume 120 of *LIPIcs*, pages 7:1–7:15, 2018.

[23] A. Brunello, D. Della Monica, and A. Montanari. Pairing monitoring with machine learning for smart system verification and predictive maintenance. In *Proceedings of the 1st Workshop on Artificial Intelligence and Formal Verification, Logic, Automata, and Synthesis (OVERLAY@AIxIA 2019)*, volume 2509, pages 71–76. CEUR-WS.org, 2019.

[24] A. Brunello, D. Della Monica, A. Montanari, N. Saccomanno, and A. Urgolo. Monitors that learn from failures: Pairing STL and genetic programming. *IEEE Access*, 11:57349–57364, 2023.

[25] A. Brunello, D. Della Monica, A. Montanari, and A. Urgolo. Learning how to monitor: Pairing monitoring and learning for online system verification. In *Proceedings of the 2nd Workshop on Artificial Intelligence and Formal Verification, Logic, Automata, and Synthesis (OVERLAY@BOSK 2020)*, volume 2785, pages 83–88. CEUR-WS.org, 2020.

[26] A. Brunello, M. Kraft, A. Montanari, F. Pittino, and A. Urgolo. Virtual sensing of temperatures in indoor environments: A case study. In *Proceedings of the 20th International Conference on Data Mining Workshops (ICDMW 2020)*, pages 805–810. IEEE, 2020.

[27] A. Brunello, G. Sciavicco, and I. E. Stan. Interval temporal logic decision tree learning. In *Proceedings of the 16th European Conference on Logics in Artificial Intelligence (JELIA 2019)*, volume 11468, pages 778–793. Springer, 2019.

[28] A. Brunello, A. Urgolo, F. Pittino, A. Montvay, and A. Montanari. Virtual sensing and sensors selection for efficient temperature monitoring in indoor environments. *Sensors*, 21(8):2728, 2021.

[29] S. Bufo, E. Bartocci, G. Sanguinetti, M. Borelli, U. Lucangelo, and L. Bortolussi. Temporal logic based monitoring of assisted ventilation in intensive care patients. In *Proceedings of the 6th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2014)*, volume 8803, pages 391–403. Springer, 2014.

[30] S. Cai, Z. Wang, S. Wang, P. Perdikaris, and G. E. Karniadakis. Physics-informed neural networks for heat transfer problems. *Journal of Heat Transfer*, 143(6), 2021.

[31] A. Calafato, C. Colombo, and G. J. Pace. A controlled natural language for tax fraud detection. In *Proceedings of the 5th International Workshop on Controlled Natural Language (CNL 2016)*, pages 1–12. Springer, 2016.

[32] A. Camacho and S. A. McIlraith. Learning interpretable models expressed in Linear Temporal Logic. In *Proceedings of the 29th International Conference on Automated Planning and Scheduling (ICAPS 2019)*, pages 621–630. AAAI Press, 2019.

[33] Y. Cao, B. J. Smucker, and T. J. Robinson. On using the hypervolume indicator to compare Pareto fronts: Applications to multi-criteria optimal experimental design. *Journal of Statistical Planning and Inference*, 160:60–74, 2015.

[34] I. Cassar and A. Francalanza. On synchronous and asynchronous monitor instrumentation for actor-based systems. *arXiv preprint arXiv:1502.03514*, 2015.

[35] I. Cassar and A. Francalanza. Runtime adaptation for actor systems. In *Proceedings of the 6th International Conference on Runtime Verification (RV 2015)*, pages 38–54. Springer, 2015.

[36] I. Cassar, A. Francalanza, L. Aceto, and A. Ingólfsdóttir. A survey of runtime monitoring instrumentation techniques. In *Proceedings of the 2nd International Workshop on Pre- and Post-Deployment Verification Techniques (PrePost@iFM 2017)*, pages 15——28. Springer, 2017.

[37] C. Cecati, G. Mokryani, A. Piccolo, and P. Siano. An overview on the smart grid concept. In *Proceedings of the 36th Annual Conference on IEEE Industrial Electronics Society (IECON 2010)*, pages 3322–3327. IEEE, 2010.

[38] G. Chandrashekar and F. Sahin. A survey on feature selection methods. *Computers & Electrical Engineering*, 40(1):16–28, 2014.

[39] O. Chapelle, B. Schölkopf, and A. Zien. *Semi-supervised learning*. MIT Press, 2006.

[40] G. Chen, M. Liu, and Z. Kong. Temporal-logic-based semantic fault diagnosis with time-series data from industrial internet of things. *IEEE Transactions on Industrial Electronics*, 68(5):4393–4403, 2020.

[41] T. Chen and C. Guestrin. XGBoost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 2016)*, pages 785–794. ACM, 2016.

[42] H. Choi, H. Kim, S. Yeom, T. Hong, K. Jeong, and J. Lee. An indoor environmental quality distribution map based on spatial interpolation methods. *Building and Environment*, 213:108880, 2022.

[43] F. Chollet et al. Keras. `https://keras.io`, 2015. Accessed: 2023-03-24.

[44] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[45] N. C. Chung, B. Miasojedow, M. Startek, and A. Gambin. Jaccard/Tanimoto similarity test and estimation methods for biological presence-absence data. *BMC Bioinformatics*, 20(15):1–11, 2019.

[46] E. M. Clarke, O. Grumberg, D. Kroening, D. A. Peled, and H. Veith. *Model checking*. MIT Press, 2nd edition, 2018.

[47] C. Da Costa, M. H. Mathias, P. Ramos, and P. S. Girão. A new approach for real time fault diagnosis in induction motors based on vibration measurement. In *Proceedings of the 27th IEEE International Instrumentation and Measurement Technology Conference (I2MTC 2010)*, pages 1164–1168. IEEE, 2010.

[48] V. K. Dabhi and S. Chaudhary. A survey on techniques of improving generalization ability of genetic programming solutions. *arXiv preprint arXiv:1211.1119*, 2012.

[49] D. de la Fuente, M. A. Vega-Rodríguez, and C. J. Pérez. Automatic selection of a single solution from the Pareto front to identify key players in social networks. *Knowledge-Based Systems*, pages 228–236, 2018.

[50] K. Deb and H. Jain. An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part I: Solving problems with box constraints. *IEEE Transactions on Evolutionary Computation*, 18(4):577–601, 2013.

[51] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-ii. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

[52] D. Della Monica and A. Francalanza. Pushing runtime verification to the limit: May process semantics be with us. In *Proceedings of the 1st Workshop on Artificial Intelligence and Formal Verification, Logic, Automata, and Synthesis (OVERLAY@AIxIA 2019)*, volume 2509, pages 47–52. CEUR-WS.org, 2019.

[53] D. Denkovski, V. Atanasovski, L. Gavrilovska, J. Riihijärvi, and P. Mähönen. Reliability of a radio environment map: Case of spatial interpolation techniques. In *Proceedings of the 7th international ICST conference on cognitive radio oriented wireless networks and communications (CROWNCOM 2012)*, pages 248–253. IEEE, 2012.

[54] A. Donzé. On signal temporal logic. In *Proceedings of the 4th International Conference on Runtime Verification (RV 2013)*, volume 8174, pages 382–383. Springer, 2013.

[55] A. Doucet, N. De Freitas, and N. Gordon. An introduction to sequential Monte Carlo methods. In *Sequential Monte Carlo Methods in Practice*, pages 3–14. Springer, 2001.

[56] C. Dousson and T. V. Duong. Discovering chronicles with numerical time constraints from alarm logs for monitoring dynamic systems. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI 1999)*, pages 620——626. Morgan Kaufmann, 1999.

[57] A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing*. Springer, 2003.

[58] B. Elhadidi and H. E. Khalifa. Application of proper orthogonal decomposition to indoor airflows. *ASHRAE Transactions*, 111(1):625–634, 2005.

[59] P. Esling and C. Agon. Time-series data mining. *ACM Computing Surveys (CSUR)*, 45(1):1–34, 2012.

[60] M. Fey and J. E. Lenssen. Fast graph representation learning with PyTorch Geometric. *arXiv preprint arXiv:1903.02428*, 2019.

[61] F. A. Fortin, F. M. De Rainville, M. A. Gardner, M. Parizeau, and C. Gagné. DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, 13(70):2171–2175, 2012.

[62] A. Francalanza, L. Aceto, and A. Ingolfsdottir. Monitorability for the Hennessy–Milner logic with recursion. *Formal Methods in System Design*, 51(1):87–116, 2017.

[63] J. H. Friedman. Greedy function approximation: A gradient boosting machine. *The Annals of Statistics*, 29(5):1189–1232, 2001.

[64] B. R. Gaines and P. Compton. Induction of ripple-down rules applied to modeling large databases. *Journal of Intelligent Information Systems*, 5(3):211–228, 1995.

[65] J. Gao, H. Wang, and H. Shen. Task failure prediction in cloud data centers using deep learning. *IEEE Transactions on Services Computing*, 15:1411–1422, 2022.

[66] M. W. Gardner and S. Dorling. Artificial neural networks (the multilayer perceptron)—a review of applications in the atmospheric sciences. *Atmospheric environment*, 32(14-15):2627–2636, 1998.

[67] M. Gerhold, A. Hartmanns, and M. Stoelinga. Model-based testing of stochastically timed systems. *Innovations in Systems and Software Engineering*, 15(3-4):207–233, 2019.

[68] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl. Neural message passing for quantum chemistry. In *Proceedings of the 34th International Conference on Machine Learning (ICML 2017)*, volume 70, pages 1263–1272. PMLR, 2017.

[69] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics (AISTATS 2010)*, volume 9, pages 249–256. JMLR.org, 2010.

[70] G. Gokhale, B. Claessens, and C. Develder. Physics informed neural networks for control oriented thermal modeling of buildings. *Applied Energy*, 314:118852, 2022.

[71] I. Gonçalves and S. Silva. Balancing learning and overfitting in genetic programming with interleaved sampling of training data. In *Proceedings of the 16th European Conference on Genetic Programming (EuroGP 2013)*, pages 73–84. Springer, 2013.

[72] I. Goodfellow, Y. Bengio, and A. Courville. *Deep learning*. MIT press, 2016.

[73] V. Gupta, T. H. Chung, B. Hassibi, and R. M. Murray. On a stochastic sensor selection algorithm with applications in sensor scheduling and sensor coverage. *Automatica*, 42(2):251–260, 2006.

[74] P. Hajihosseini, M. M. Anzehaee, and B. Behnam. Fault detection and isolation in the challenging Tennessee Eastman Process by using image processing techniques. *ISA Transactions*, 79:137–146, 2018.

[75] W. L. Hamilton. Graph representation learning. *Synthesis Lectures on Artifical Intelligence and Machine Learning*, 14(3):1–159, 2020.

[76] J. Z. Hare, S. Gupta, and T. A. Wettergren. POSE: Prediction-based opportunistic sensing for energy efficiency in sensor networks using distributed supervisors. *IEEE Transactions on Cybernetics*, 48(7):2114–2127, 2017.

[77] K. Havelund and D. Peled. Runtime verification: from propositional to first-order temporal logic. In *Proceedings of the 18th International Conference on Runtime Verification (RV 2018)*, pages 90–112. Springer, 2018.

[78] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

[79] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[80] S. Holm. A simple sequentially rejective multiple test procedure. *Scandinavian journal of statistics*, 6(2):65–70, 1979.

[81] K. Hornik. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2):251–257, 1991.

[82] X. Huang. *Hard drive failure prediction for large scale storage system*. PhD thesis, University of California, Los Angeles, 2017.

[83] S. R. Islam, D. Kwak, M. H. Kabir, M. Hossain, and K.-S. Kwak. The internet of things for health care: a comprehensive survey. *IEEE access*, 3:678–708, 2015.

[84] G. James, D. Witten, T. Hastie, and R. Tibshirani. *An introduction to statistical learning*, volume 112. Springer, 2013.

[85] N. Jansen, J.-P. Katoen, P. Kohli, and J. Kretinsky. Machine learning and model checking join forces. *Dagstuhl Reports*, 8(3):74–93, 2018.

[86] S. T. Jawaid and S. L. Smith. Submodularity and greedy algorithms in sensor scheduling for linear dynamical systems. *Automatica*, 61:282–288, 2015.

[87] H. Jin, L. Su, D. Chen, K. Nahrstedt, and J. Xu. Quality of information aware incentive mechanisms for mobile crowd sensing systems. In *Proceedings of the 16th ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc 2015)*, pages 167–176. ACM, 2015.

[88] D. Kasenberg and M. Scheutz. Interpretable apprenticeship learning with temporal logic specifications. In *Proceedings of the 56th IEEE Conference on Decision and Control (CDC 2017)*, pages 4914–4921. IEEE, 2017.

[89] M. Kashiwagi, C. da Costa, and M. Mathias. Development of diagnosis system for rolling bearings faults on real time based on FPGA. *Renewable Energy and Power Quality Journal*, 10(10):545–549, 2012.

[90] E. Keogh, S. Lonardi, C. A. Ratanamahatana, L. Wei, S.-H. Lee, and J. Handley. Compression-based data mining of sequential data. *Data Mining and Knowledge Discovery*, 14(1):99–129, 2007.

[91] D. Kim, S. C. Han, Y. Lin, B. H. Kang, and S. Lee. RDR-based knowledge based system to the failure detection in industrial cyber physical systems. *Knowledge-Based Systems*, 150:1–13, 2018.

[92] J. Kim, C. Muise, A. Shah, S. Agarwal, and J. Shah. Bayesian inference of Linear Temporal Logic specifications for contrastive explanations. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI 2019)*, pages 5591–5598. IJCAI Organization, 2019.

[93] T. S. Kim and S. Y. Sohn. Multitask learning for health condition identification and remaining useful life prediction: Deep convolutional neural network approach. *Journal of Intelligent Manufacturing*, 32(8):2169–2179, 2020.

[94] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[95] S. C. Kleene. *Introduction to metamathematics*. Princeton, NJ, 1952.

[96] R. Koenker and K. F. Hallock. Quantile regression. *Journal of Economic Perspectives*, 15(4):143–156, 2001.

[97] P. Korvesis, S. Besseau, and M. Vazirgiannis. Predictive maintenance in aviation: Failure prediction from post-flight reports. In *Proceedings of the 34th IEEE International Conference on Data Engineering (ICDE 2018)*, pages 1414——1422. IEEE, 2018.

[98] J. R. Koza. Genetic programming as a means for programming computers by natural selection. *Statistics and Computing*, 4(2):87–112, 1994.

[99] S. K. Kumar. On weight initialization in deep neural networks. *arXiv preprint arXiv:1704.08863*, 2017.

[100] M. Lambardi di San Miniato, R. Bellio, L. Grassetti, and P. Vidoni. Separable spatio-temporal kriging for fast virtual sensing. *Applied Stochastic Models in Business and Industry*, 38(5):806–829, 2022.

[101] D. J. Lary, A. H. Alavi, A. H. Gandomi, and A. L. Walker. Machine learning in geosciences and remote sensing. *Geoscience Frontiers*, 7(1):3–10, 2016.

[102] D. J. Lary, F. S. Faruque, N. Malakar, A. Moore, B. Roscoe, Z. L. Adams, and Y. Eggelston. Estimating the global abundance of ground level presence of particulate matter (pm2.5). *Geospatial Health*, 8(3):611, 2014.

[103] T. D. B. Le and D. Lo. Deep specification mining. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018)*, pages 106——117. ACM, 2018.

[104] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, volume 86, pages 2278–2324. IEEE, 1998.

[105] C. Lemieux, D. Park, and I. Beschastnikh. General LTL specification mining (T). In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE 2015)*, pages 81–92. IEEE/ACM, 2015.

[106] M. Leucker and C. Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293–303, 2009.

[107] H. Li, D. Yu, and J. E. Braun. A review of virtual sensing technology and application in building systems. *HVAC&R Research*, 17(5):619–645, 2011.

[108] J. Lin, E. Keogh, L. Wei, and S. Lonardi. Experiencing SAX: A novel symbolic representation of time series. *Data Mining and Knowledge Discovery*, 15(2):107–144, 2007.

[109] L. Liu, S. M. Kuo, and M. Zhou. Virtual sensing techniques and their applications. In *Proceedings of the 2009 International Conference on Networking, Sensing and Control (ICNSC 2009)*, pages 31–36. IEEE, 2009.

[110] S. Lu, B. Luo, T. Patel, Y. Yao, D. Tiwari, and W. Shi. Making disk failure predictions SMARTer! In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST 2020)*, pages 151–167. USENIX Association, 2020.

[111] S. Luke and L. Panait. Fighting bloat with nonparametric parsimony pressure. In *Proceedings of the 7th International Conference on Parallel Problem Solving from Nature (PPSN 2002)*, volume 2439, pages 411——421. Springer, 2002.

[112] S. M. Lundberg and S.-I. Lee. A unified approach to interpreting model predictions. In *Advances in Neural Information Processing Systems 30*, pages 4765–4774. Curran Associates, Inc., 2017.

[113] J. Ma, Y. Ding, J. C. Cheng, F. Jiang, and Z. Wan. A temporal-spatial interpolation and extrapolation method based on geographic long short-term memory neural network for pm2.5. *Journal of Cleaner Production*, 237:117729, 2019.

[114] R. Ma, N. Liu, X. Xu, Y. Wang, H. Y. Noh, P. Zhang, and L. Zhang. Fine-grained air pollution inference with mobile sensing systems: A weather-related deep autoencoder model. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 4(2):1–21, 2020.

[115] M. S. Mahdavinejad, M. Rezvan, M. Barekatain, P. Adibi, P. Barnaghi, and A. P. Sheth. Machine learning for internet of things data analysis: A survey. *Digital Communications and Networks*, 4(3):161–175, 2018.

[116] O. Maler and D. Nickovic. Monitoring temporal properties of continuous signals. In *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, pages 152–166. Springer, 2004.

[117] H. B. Mann and D. R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The Annals of Mathematical Statistics*, 18(1):50–60, 1947.

[118] Z. Manna and A. Pnueli. *Temporal verification of reactive systems - safety.* Springer, 1995.

[119] F. Manzella, G. Pagliarini, G. Sciavicco, and I. E. Stan. The voice of COVID-19: Breath and Cough Recording Classification with Temporal Decision Trees and Random Forests. *Artificial Intelligence in Medicine*, 137:102486, 2023.

[120] M. Martínez-Comesaña, A. Ogando-Martínez, F. Troncoso-Pastoriza, J. López-Gómez, L. Febrero-Garrido, and E. Granada-Álvarez. Use of optimised MLP neural networks for spatiotemporal estimation of indoor environmental conditions of existing buildings. *Building and Environment*, 205:108243, 2021.

[121] F. J. Massey. The kolmogorov-smirnov test for goodness of fit. *Journal of the American Statistical Association*, 46(253):68–78, 1951.

[122] T. Mitsa. *Temporal data mining*. Chapman and Hall/CRC, 2010.

[123] Y. Mo, R. Ambrosino, and B. Sinopoli. Sensor selection strategies for state estimation in energy constrained wireless sensor networks. *Automatica*, 47(7):1330–1338, 2011.

[124] M. Mohammadi, A. Al-Fuqaha, S. Sorour, and M. Guizani. Deep learning for iot big data and streaming analytics: A survey. *IEEE Communications Surveys & Tutorials*, 20(4):2923–2960, 2018.

[125] S. Mohammadinejad, J. V. Deshmukh, A. G. Puranic, M. Vazquez-Chanlatte, and A. Donzé. Interpretable classification of time-series data using efficient enumerative techniques. In *Proceedings of the 23rd ACM International Conference on Hybrid Systems: Computation and Control (HSCC 2020)*, pages 1–10. ACM, 2020.

[126] B. Mohammed, I. Awan, H. Ugail, and M. Younas. Failure prediction using machine learning in a virtualised HPC system and application. *Cluster Computing*, 22(2):471–485, 2019.

[127] C. Montzka, H. Moradkhani, L. Weihermüller, and H.-J. H. Franssen. Hydraulic parameter estimation by remotely-sensed top soil moisture observations with the particle filter. *Journal of Hydrology*, 399(3-4):410–421, 2011.

[128] A. A. Motsinger, S. L. Lee, G. Mellick, and M. D. Ritchie. Gpnn: Power studies and applications of a neural network method for detecting gene-gene interactions in studies of human disease. *BMC bioinformatics*, 7(1):1–10, 2006.

[129] G. J. Myers, C. Sandler, and T. Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[130] N. Najjar, S. Gupta, J. Hare, S. Kandil, and R. Walthall. Optimal sensor selection and fusion for heat exchanger fouling diagnosis in aerospace systems. *IEEE Sensors Journal*, 16(12):4866–4881, 2016.

[131] P. Naldurg, K. Sen, and P. Thati. A temporal logic based framework for intrusion detection. In *Proceedings of the 24th International Conference on Formal Techniques for Networked and Distributed Systems (FORTE 2004)*, volume 3235, pages 359–376. Springer, 2004.

[132] D. Neider and I. Gavran. Learning linear temporal properties. In *Proceedings of the 18th Formal Methods in Computer Aided Design (FMCAD 2018)*, pages 1–10. IEEE, 2018.

[133] L. Nenzi, S. Silvetti, E. Bartocci, and L. Bortolussi. A robust genetic algorithm for learning temporal specifications from data. In *Proceedings of the 15th International Conference on Quantitative Evaluation of Systems (QEST 2018)*, volume 11024, pages 323–338. Springer, 2018.

[134] Y. Nesterov. A method for unconstrained convex minimization problem with the rate of convergence O(1/kˆ2). In *Doklady AN USSR*, volume 269, pages 543–547, 1983.

[135] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6):89–100, 2007.

[136] D. Ničković and T. Yamaguchi. RTAMT: Online robustness monitors from STL. In *Proceedings of the 18th International Symposium on Automated Technology for Verification and Analysis (ATVA 2020)*, volume 12302, pages 564—–571. Springer, 2020.

[137] C. Nwankpa, W. Ijomah, A. Gachagan, and S. Marshall. Activation functions: Comparison of trends in practice and research for deep learning. *arXiv preprint arXiv:1811.03378*, 2018.

[138] E. Oktavia, Widyawan, and I. W. Mustika. Inverse distance weighting and kriging spatial interpolation for data center thermal monitoring. In *Proceedings of the 1st International Conference on Information Technology, Information Systems and Electrical Engineering (ICITISEE 2016)*, pages 69–74. IEEE, 2016.

[139] M. Onel, C. A. Kieslich, and E. N. Pistikopoulos. A nonlinear support vector machine-based feature selection approach for fault detection and diagnosis: Application to the Tennessee Eastman Process. *American Institute of Chemical Engineers Journal*, 65(3):992–1005, 2019.

[140] S. Oswal, A. Singh, and K. Kumari. Deflate compression algorithm. *International Journal of Engineering Research and General Science*, 4(1):430–436, 2016.

[141] K. A. Palmer and G. M. Bollas. Sensor selection embedded in active fault diagnosis algorithms. *IEEE Transactions on Control Systems Technology*, 2019.

[142] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Proceedings of the 33rd Conference on Neural Information Processing Systems (NeurIPS 2019)*, volume 32, pages 8026–8037. Curran Associates, Inc., 2019.

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

[144] G. Petmezas, K. Haris, L. Stefanopoulos, V. Kilintzis, A. Tzavelis, J. A. Rogers, A. K. Katsaggelos, and N. Maglaveras. Automated atrial fibrillation detection using a hybrid CNN-LSTM network on imbalanced ECG datasets. *Biomedical Signal Processing and Control*, 63:102194, 2021.

[145] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science (FOCS 1977)*, pages 46–57. IEEE, 1977.

[146] R. Poli, W. Langdon, and N. Mcphee. *A field guide to genetic programming*. http://www.gp-field-guide.org.uk, 2008.

[147] G. Priest. The logic of paradox. *Journal of Philosophical logic*, 8(1):219–241, 1979.

[148] Property pattern mappings for LTL, kansas state university cis department, Laboratory for Specification, Analysis, and Transformation of Software (SAnToS Laboratory). `https://matthewbdwyer.github.io/psp/`. Accessed: 2023-03-24.

[149] J. R. Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[150] J. R. Quinlan. Simplifying decision trees. *International journal of man-machine studies*, 27(3):221–234, 1987.

[151] J. R. Quinlan. *C4. 5: programs for machine learning*. Morgan Kaufmann Publishers Inc., 1993.

[152] M. Raissi, P. Perdikaris, and G. Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, 378:686–707, 2019.

[153] E. Ramasso and A. Saxena. Performance benchmarking and analysis of prognostic methods for CMAPSS datasets. *International Journal of Prognostics and Health Management*, 5:1–15, 11 2014.

[154] M. T. Ribeiro, S. Singh, and C. Guestrin. "Why should i trust you?" Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining (KDD 2016)*, pages 1135–1144, 2016.

[155] C. A. Rieth, B. D. Amsel, R. Tran, and M. B. Cook. Issues and advances in anomaly detection evaluation for joint human-automated systems. In *Proceedings of the 8th International Conference on Human Factors in Robots and Unmanned Systems (AHFE 2017)*, pages 52–63. Springer, Springer, 2017.

[156] J. P. Rigol, C. H. Jarvis, and N. Stuart. Artificial neural networks as a tool for spatial interpolation. *International Journal of Geographical Information Science*, 15(4):323–343, 2001.

[157] A. Rodionova, E. Bartocci, D. Nickovic, and R. Grosu. Temporal logic as filtering. In *Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control (HSCC 2016)*, pages 11——20. ACM, 2016.

[158] L. Rokach and O. Maimon. Top-down induction of decision trees classifiers-a survey. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 35(4):476–487, 2005.

[159] P. Rombach and J. Keuper. SmartPred: Unsupervised hard disk failure detection. In *Proceedings of the 35th International Conference on High Performance Computing (ISC High Performance 2020)*, volume 12321, pages 235–246. Springer, 2020.

[160] S. Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[161] D. E. Rumelhart, G. E. Hinton, J. L. McClelland, et al. A general framework for parallel distributed processing. *Parallel distributed processing: Explorations in the microstructure of cognition*, 1(45-76):26, 1986.

[162] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[163] R. Saheba, M. Rotea, O. Wasynczuk, S. Pekarek, and B. Jordan. Virtual thermal sensing for electric machines. *IEEE Control Systems Magazine*, 30(1):42–56, 2010.

[164] A. Sanchez-Gonzalez, J. Godwin, T. Pfaff, R. Ying, J. Leskovec, and P. Battaglia. Learning to simulate complex physics with graph networks. In *Proceedings of the 37th International Conference on Machine Learning (ICML 2020)*, volume 119, pages 8459–8468. PMLR, 2020.

[165] V. Satopaa, J. Albrecht, D. Irwin, and B. Raghavan. Finding a ''kneedle'' in a haystack: Detecting knee points in system behavior. In *Proceedings of the 31st International Conference on Distributed Computing Systems (ICDCS 2011) Workshops*, pages 166–171. IEEE, 2011.

[166] A. Saxena, K. Goebel, D. Simon, and N. Eklund. Damage propagation modeling for aircraft engine run-to-failure simulation. In *Proceedings of the 1st International Conference on Prognostics and Health Management (PHM 2008)*, pages 1–9. IEEE, 2008.

[167] J. L. Schafer and J. W. Graham. Missing data: our view of the state of the art. *Psychological methods*, 7(2):147, 2002.

[168] M. Schuster and K. K. Paliwal. Bidirectional recurrent neural networks. *IEEE transactions on Signal Processing*, 45(11):2673–2681, 1997.

[169] G. Sciavicco, I. E. Stan, and A. Vaccari. Towards a general method for logical rule extraction from time series. In *Proceedings of the 8th International Work-Conference on the Interplay Between Natural and Artificial Computation (IWINAC 2019)*, volume 11487 of *Lecture Notes in Computer Science*, pages 3–12, 2019.

[170] A. Sempey, C. Inard, C. Ghiaus, and C. Allery. Fast simulation of temperature distribution in air conditioned rooms by using proper orthogonal decomposition. *Building and Environment*, 44(2):280–289, 2009.

[171] A. J. Shah, P. Kamath, J. A. Shah, and S. Li. Bayesian inference of temporal task specifications from demonstrations. In *Proceedings of the 32nd Conference on Neural Information Processing Systems (NeurIPS 2018)*, pages 3808–3817. NeurIPS Foundation, 2018.

[172] S. E. Snell, S. Gopal, and R. K. Kaufmann. Spatial interpolation of surface air temperatures using artificial neural networks: Evaluating their use for downscaling gcms. *Journal of Climate*, 13(5):886–895, 2000.

[173] C.-J. Su and Y. Li. Recurrent neural network based real-time failure detection of storage devices. *Microsystem Technologies*, 28(2):621–633, 2019.

[174] S. Vallachira, M. Orkisz, M. Norrlöf, and S. Butail. Data-driven gearbox failure detection in industrial robots. *IEEE Transactions on Industrial Informatics*, 16(1):193–201, 2019.

[175] V. Vapnik. *Statistical learning theory*. Wiley, 1998.

[176] P. Velickovic, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

[177] C. Venkatesh, N. Rengarajan, P. Ponmurugan, and S. Balamurugan. *Smart Systems for Industrial Applications*. John Wiley & Sons, 2022.

[178] M. P. Vitus, W. Zhang, A. Abate, J. Hu, and C. J. Tomlin. On efficient sensor scheduling for linear dynamical systems. *Automatica*, 48(10):2482–2493, 2012.

[179] J. Wang, J. Xie, R. Zhao, L. Zhang, and L. Duan. Multisensory fusion based virtual tool wear sensing for ubiquitous manufacturing. *Robotics and Computer-Integrated Manufacturing*, 45:47–58, 2017.

[180] J. Wang, Y. Zheng, P. Wang, and R. X. Gao. A virtual sensing based augmented particle filter for tool condition prognosis. *Journal of Manufacturing Processes*, 28:472–478, 2017.

[181] Y. Watanabe, H. Otsuka, M. Sonoda, S. Kikuchi, and Y. Matsumoto. Online failure prediction in cloud datacenters by real-time message pattern learning. In *Proceedings of the 4th IEEE International Conference on Cloud Computing Technology and Science (CloudCom 2012)*, pages 504–511. IEEE, 2012.

[182] P. J. Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

[183] F. Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80–83, 1945.

[184] R. J. Williams and D. Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, 1(2):270–280, 1989.

[185] J. Wu, Q.-S. Jia, K. H. Johansson, and L. Shi. Event-based sensor data scheduling: Trade-off between communication rate and estimation quality. *IEEE Transactions on Automatic Control*, 58(4):1041–1046, 2012.

[186] C. Xu, H. Chen, J. Wang, Y. Guo, and Y. Yuan. Improving prediction performance for indoor temperature in public buildings based on a novel deep learning method. *Building and Environment*, 148:128–135, 2019.

[187] H. Xue, W. Jiang, C. Miao, Y. Yuan, F. Ma, X. Ma, Y. Wang, S. Yao, W. Xu, A. Zhang, and other. DeepFusion: A deep learning framework for the fusion of heterogeneous sensory data. In *Proceedings of the 20th ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc 2019)*, pages 151–160. ACM, 2019.

[188] S. Yang, X. Liu, S. Liu, X. Chen, and Y. Cao. Real-time temperature distribution monitoring in chinese solar greenhouse using virtual lan. *Agronomy*, 12(7):1565, 2022.

[189] S. Yao, Y. Zhao, A. Zhang, S. Hu, H. Shao, C. Zhang, L. Su, and T. Abdelzaher. Deep learning for the internet of things. *Computer*, 51(5):32–41, 2018.

[190] Z. Yu, Y. Song, D. Song, and Y. Liu. Spatial interpolation-based analysis method targeting visualization of the indoor thermal environment. *Building and Environment*, 188:107484, 2021.

[191] A. Zheng and A. Casari. *Feature engineering for machine learning: principles and techniques for data scientists*. O'Reilly Media, Inc., 2018.

[192] Towards a vision of innovative smart systems integration - european platform on smart systems integration, eposs. `https://www.smart-systems-integration.org/ssi-smart-systems-integration`. Accessed: 2023-03-24.

[193] Use cases and benefits of smart sensors for iot. `https://internetofthingsagenda.techtarget.com/opinion/`

How-smart-sensors-are-transforming-the-Internet-of-Things. Accessed: 2023-03-24.