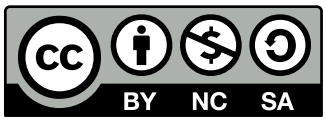
Aalto University
School of Science

Master's Programme in ICT-Innovation

Efficient Online Learning in Resource-Constrained Automation Environments

Marco Di Francesco

Master's Thesis
2024



Aalto University
School of Science

Author Marco Di Francesco

Title Efficient Online Learning in Resource-Constrained Automation Environments

Degree programme ICT-Innovation

Major Data Science

Supervisor Dr.-Ing. Kuan-Hsun Chen

Advisor Jordis Herrmann

Collaborative partner ABB

Date 16 August 2024 **Number of pages** 45+9 **Language** English

Abstract

Machine learning solutions have proven highly effective for various tasks in recent years. However, their use in an automation environment requires that they run locally with limited resources, in a setting called Edge Computing. At the same time, there is a need to facilitate continuous improvements and updates throughout the product lifecycle to ensure that systems are adaptable to evolving environments and under performance degradation of machines. For this reason, Incremental Learning models have become increasingly relevant due to their ability to process data in real-time, while also lifting the need to store all data in memory. However, efficiency in these models is often overlooked, with many implementations in Python resulting in a substantial memory footprint and slow execution, making the usage of such models in robotic controllers impractical due to the high cost of improving hardware. In this work, we implement an efficient online learning model called Mondrian Forests using the Rust language, achieving a 28-fold improvement in execution speed compared to the Python implementation. Additionally, we apply memory optimizations through spatial locality caching, further reducing execution time by 18%. Consequently, we measure performance using datasets from real-world industrial settings, analyzing the implications for automation.

Keywords edge computing, incremental learning, model efficiency, spatial locality caching

Contents

1 Introduction

Over recent years, machine learning solutions have proven highly effective for a wide range of tasks. However, deploying them in an automation environment requires that they not only run locally with limited computational resources but also facilitate continuous improvements and updates throughout the product's lifecycle. In such settings, what is normal for one robot may be an anomaly for another. Therefore, it is important to learn the normal behavior of each individual robot to detect deviations effectively. This adaptability is needed to adjust to changes such as the replacement of a spare part, or relocation to a different task. Given that typical machine offline learning models in an automation setting cannot adapt to ever-changing environments, online learning models have become increasingly relevant due to their ability to continuously adapt to the environment.

In addition to adaptability, real-time scenarios demand that the learned model deployed on edge computing devices be executed efficiently. Consider an illustrative setup where robotic controllers in an automated manufacturing plant monitor and control welding processes. These controllers, operating with limited computational resources and energy constraints, must process sensor data in real time to ensure precise operations and quick responses to anomalies. These tasks already consume a significant portion of the available computational resources, given the numerous simultaneous processes running. To achieve low latency and ensure privacy, the robotic controllers apply machine learning models directly to the raw sensor data. By optimizing the execution of these machine learning models, we significantly reduce computational resource consumption, thereby lowering the hardware requirements for the robot controllers and enabling cost savings.

Given these hardware constraints, decision trees have become a popular choice for machine learning models in such environments due to their computational efficiency on CPUs and fast inference capabilities. Other solutions for these tasks include deep neural networks, which have inherently deep structures requiring extensive calculations during the training phase, thus exhibiting significantly longer execution times per iteration. In contrast, decision trees offer a much faster solution for real-time requirements. Furthermore, while Reinforcement Learning (RL) is inherently suited for real-time and online learning tasks, its application in small computing environments is limited due to significant memory requirements. For instance, the survey by Kober et al. [16] highlights various applications of RL in robotic tasks and notes that one significant limitation of these models is their substantial memory requirements, rendering them impractical for deployment on resource-constrained robotic controllers.

However, the current implementations of online decision tree models present several challenges. At the time of writing, all implementations are in Python, which, like other interpreted languages, introduces substantial computational overhead. This makes it computationally expensive to execute interpreted code on the resource-constrained controllers targeted by this project, thus conflicting with the minimal resource requirements of an automation environment. Moreover, in a production context with many processes running on the same controller, it is important to ensure that the system does not run out of memory, which current implementations cannot

guarantee. Python's language abstraction and the complexity of its data structures, along with the libraries built on top of it, make it difficult to precisely measure memory usage. Technologies that do not rely directly on Python have been proposed. For instance, TensorFlow Lite [1], a cross-platform framework for efficient model deployment, and ONNX [4], an open-source format for model interoperability and deployment, are designed for inference on edge devices. However, these tools are not suitable for training or further tuning the model in an incremental setting.

For these reasons, the implementation of these models in a non-interpreted language is necessary. The choice of low-level language involves a trade-off between execution speed, memory efficiency, and ease of implementation. In this regard, the work from Pereira *et al.* [24] studies the overview of different programming languages. The global implementation of their tests has shown that the languages with the fastest execution are C, Rust, and C++, with a remarkable difference in performance compared to Python, being about 45 to 70 times faster. Among these languages, Rust is the best candidate due to its combination of performance, memory safety, and an existing online machine learning library, LightRiver [2]. This library facilitates the implementation of online learning models by allowing developers to build on top of existing data structures and project frameworks.

As mentioned before, the robotic environment we are working with requires using resources as sparingly as possible, and better-performing models would translate to lower hardware requirements and reduced costs. In this regard, the work proposed by Kuan et al. [6] aims to speed up model execution by applying optimizations with spatial locality in an inference context. The optimizations in their work have been applied to models in offline learning methods, but none have extended these techniques to online learning settings. In this work, we build upon the foundational work proposed by Kuan et al. by applying their model optimization techniques to online learning scenarios, thus enabling real-time adaptation and performance improvements in dynamic environments.

Summary of Our Contributions:

- We implement the Mondrian Forest model in Rust for classification and regression tasks.
- We apply cache-aware optimization to the online learning model.
- We study the correlation between the speedup achieved with optimizations and the structural properties of the tree.
- We calculate the cost of applying the optimizations and the gain we receive from them.
- We evaluate the developed model by testing it on two datasets with the industrial environment, and analyzing its performance.
- We investigate techniques for model reduction and limitation to ensure efficient performance on resource-constrained devices.

The implementation and results of this work, including all the developed code, experiments, and analyses, are made accessible in the repository https://github.com/MarcoDiFrancesco/light-river-cache.

Research questions:

- **RQ1:** *How does vector sorting optimization on the online implementation of Mondrian Forest affect the model's execution time?* This research question investigates whether the vector sorting optimization technique, previously applied in offline methods, can enhance the execution speed of the Mondrian Forest model in an online learning context.

- **RQ2:** *What is the optimal frequency for applying optimizations?* This question aims to determine the most effective frequency for applying optimizations in an online learning environment, balancing the time required for optimization with the resulting performance improvements.

- **RQ3:** *How does the number of trees impact the memory footprint of the model in a robotic controller?* This question examines the relationship between the number of trees in the model and its memory usage, providing insights for future scalability by estimating memory consumption and adjusting hyperparameters to maintain accuracy while adhering to the memory constraints of various controllers.

In this work, we explore the background in Section 2, focusing on edge computing and the associated hardware and learning strategies. We then examine decision tree models for the edge, comparing various decision tree variants and their suitability for edge environments. Additionally, we discuss optimization techniques, including cache-aware optimization and array mapping, to enhance model performance. Finally, we discuss the different types of drift. Section 3 covers the methodology, including the implementation of the Mondrian Forest model in Rust, the application of optimizations to this model, and the evaluation of these models. Section 4 presents the results, demonstrating significant improvements in execution time and memory usage with our implementation in Rust, as well as the outcomes of the optimizations and their evaluation on two industrial datasets. In Section 5, we discuss the broader implications of our findings. Finally, in Section 6, we suggest directions for future research, including further optimizations and the development of base models for diverse robotic environments.

2 Background

2.1 What is the Edge?

In this section, we explore Edge Computing (EC) and Edge Intelligence (EI) concepts within the context of robotic systems, focusing on defining the hardware choices integral to these fields. Additionally, we examine the learning strategies applicable in Edge Computing environments, highlighting their advantages, challenges, and practical implications in robotics.

2.1.1 Edge Definitions

Edge Computing Edge Computing (EC) refers to a distributed computing paradigm that brings computation and data processing closer to the location where it is needed [26]. This paradigm is related to the concept of the Internet of Things (IoT), which primarily focuses on data collection. In contrast, EC includes computing capabilities in the device. The EC approach brings several major benefits, with varying levels of importance. In the context of automation, the main benefits are ranked in order of importance as follows:

- Latency: The primary goal of EC is to achieve faster response times by placing computational resources closer to data sources. High latency in robotic systems is not acceptable, as it hinders real-time decision-making. Local data processing mitigates this issue, allowing robots to make real-time predictions with minimal delay.
- Privacy: EC significantly reduces the transmission of sensitive information over networks by facilitating local data processing. This is particularly important in the commercial automation sector, where clients are often against transmitting robot data that may contain insights from their intellectual property.
- Reliability: System reliability is enhanced by allowing localized data processing to continue even if connectivity to central servers is lost or irregular. In the automation context, this is important since downtimes are costly.
- Cost: Robotic systems utilizing EC can achieve significant cost savings by lowering operational costs for data processing and storage through localized data processing.
- Energy Efficiency: By reducing the need for long-distance data transmission and central processing, EC can contribute to overall energy efficiency.
- Bandwidth: EC reduces the load on network bandwidth by processing data locally. This minimizes the quantity of data sent to central data centers. In the robotics context, sending thousands of data points per second from sensors to a centralized location can be resource-intensive.

Edge Intelligence Edge Intelligence (EI), also known as TinyML [28], is a subset of EC that specifically involves the integration of Artificial Intelligence (AI) with EC systems. EI expands on the idea of EC by integrating AI algorithms and machine learning models at the network's edge. However, it is important to note that EI necessitates computational processing at the network edge, i.e., devices like IoT security cameras employing computer vision that require powerful external GPUs for processing do not align with this definition, as their operational reliance is primarily on cloud-based cognitive services [32].

When developing EI systems, in addition to EC variables, we should take into consideration factors related to the integration of AI and ML models:

- Memory: In an EI setting, computational resources are constrained, thus optimizing memory usage by choosing lightweight ML models should be done, possibly without compromising processing capability or accuracy.
- Accuracy: EI necessitates assessing the accuracy and reliability of AI algorithms. This involves considering the potential effects of model accuracy on the edge system's overall effectiveness.

Recent research by Zhang *et al.* in this field highlights the need for more research into the trade-off between memory and latency as an open problem in the model inference space [32]. This is still an area of focus for research, requiring examination to maximize the trade-off between quick response times (low latency) and effective memory use in computational models.

2.1.2 Edge Hardware

In the context of Edge Computing system development, it is important to consider the architecture of the underlying computing units when developing new algorithms. This classification defines two very different approaches used in the literature: reconfigurable and fixed instruction set architecture computing units.

Reconfigurable units Reconfigurable computing hardware, such as Field Programmable Gate Arrays (FPGAs), can be reconfigured after manufacturing to perform specific tasks, making them adaptable for various specialized computational needs [7]. Application-Specific Integrated Circuits (ASICs) represent a specialized form within this category. Since ASICs are specifically made for a given task they can provide even a higher efficiency compared to FPGAs [7]. ASICs are generally more expensive to develop but once mass-produced they can have an overall lower price per unit. Due to their highly effective computing capabilities, reconfigurable computing units like FPGAs have seen an increase in the adoption of EI [25].

Fixed-ISA units Opposed to reconfigurable computing units are units with a fixed instruction set architecture (ISA). The ISA of a computing machine includes the definition of the fixed binary instructions, registers, and memory space usable by

the unit [29], as opposed to the fully customizable instructions of the reconfigurable computing hardware units. These kinds of processing units are intended for general-purpose computing tasks and are constructed on fixed hardware architectures. An example of such are ARM and Intel processors.

There are two main architectures in this category: reduced and complex ISAs [27]. Reduced Instruction Set Computer (RISC) architectures, such as ARM and RISC-V, focus on a reduced set of instructions to optimize performance and simplify operations. Since it requires less power and allows for faster processing speeds it is ideal for edge devices. Complex instruction set computer (CISC) architectures like x86 and x86_64 are frequently seen in Intel processors and use an increased number of instructions to carry out a wider range of advanced operations. Both architectures have evolved with time and can now be applied in EC environments with different computational requirements.

2.1.3 Edge Learning Strategies

In this section, the analysis of Edge Learning Strategies of Machine Learning models in Edge Computing environments is conducted. This analysis includes different learning methods that can be categorized as offline, transfer, incremental, online, and federated.

Offline Learning Traditional deployment of pre-trained models offers simplicity, low computational overhead, and smaller hardware requirements when deploying to the edge. These models can be rapidly deployed without ongoing training or updates, making them ideal for tasks requiring consistent performance and low-latency responses. One key advantage of offline learning is that models can be verified and tested before deployment. However, updating offline models requires access to the entire dataset during the training phase, incurring significant computational costs that edge devices cannot handle. In automation scenarios, manufacturers may update such models at fixed intervals, such as annually.

Transfer Learning As explained in the survey by Zhuang *et al.* [33], transfer learning involves deploying a pre-trained model, which has been developed on a comprehensive dataset, and then adapting it to the specific characteristics of the automation environment. This adaptation uses the computational efficiency of the pre-trained model while finetuning it to the unique requirements of the task. An example in the automation context could be training the model with data coming from many robots, and later finetuning it on a specific robot's dataset to fit its environment.

Incremental Learning As defined by Losing *et al.* [19], incremental learning represents an approach in which a machine learning model is continuously updated as it receives new data, rather than being trained once on a fixed dataset. Unlike transfer learning, where a pre-trained model is adapted to a new task only once, incremental learning involves the model adapting to new patterns and information over time, while still retaining previously learned knowledge. The model integrates new data patterns

while maintaining previously acquired knowledge, with the balance between new and old data being governed by parameters that can control the update rate. Incremental learning is beneficial in edge computing contexts due to its ability to locally update models on edge devices with incoming data, without the need for data transfer to central servers like in the previous techniques.

Online Learning Defined by Losing *et al.* [19], online learning refers to a strategy subset of incremental learning. Unlike incremental learning, online learning is specifically focused on adapting to new data on the fly by incrementally updating its parameters in response to each new data point. This strategy is pertinent in edge computing scenarios where data is generated in a continuous stream and requires immediate processing. In this context, the concept of *anytime prediction* describes how the model can provide valid output at any point during its operation, progressively refining the prediction as more data is processed. Key challenges in this approach include maintaining model stability during continuous updates and dealing with potentially non-stationary data distributions, known as drift, later explained in Section 2.4. The main challenge of current online learning algorithms is the necessity for a larger volume of training data to retain the same predictive performance as their batch-processed counterparts [17].

Federated Learning First introduced by McMahan *et al.* [21], federated learning involves a decentralized machine learning paradigm where model training occurs across a network of distributed devices, each with its local data. Each edge device independently computes model updates based on its local data, contributing to a global model through a process of periodic aggregation on a central server. The implementation of this method requires efficient coordination and communication strategies to address the challenges posed by the heterogeneity of data and computational resources. The main challenges include managing asynchronous model updates, ensuring global model convergence despite data not being Independent and Identically Distributed (IID), and optimizing communication protocols to mitigate latency. Moreover, in commercial robotics environments, this approach avoids the issue of clients having to send potentially sensitive or intellectual property-related data, which is a significant barrier to training centralized models.

2.2 Models for the Edge

This section explores a machine learning model suitable for incremental learning tasks. Initially, we discuss the rationale behind selecting decision trees for our domain, followed by a comparison of different variants of this model.

Decision Tree A Decision Tree (DT) is a fundamental machine learning model extensively used in both regression and classification problems. Characterized by its tree-like structure, a decision tree splits the data into subsets based on the values of input features, creating branches based on decisions to reduce the discrepancy between

the input and the prediction. Decision trees can handle both numerical and categorical data, and they are capable of modeling complex, non-linear relationships.

One major difference between decision tree models and other machine learning models is their ability to exhibit faster training times on CPUs as opposed to GPUs, which means they are well suited for CPU-only settings like in Edge Intelligence. The reason behind this characteristic is primarily given by the difficulty involved in the parallelization of the DT model. The tree-like structure of decision trees impedes their natural decomposition into equally sized tasks suitable for parallel processing on GPUs. This structural characteristic leads to inefficiencies in the distribution of tasks and data across GPU cores, as well as in the aggregation of results, thereby diminishing the advantages of parallelization.

More specifically, as highlighted by Zhang *et al.* [31], the major computational expense in training decision tree ensembles is attributed to the training of individual trees. The challenge lies in identifying the optimal split for each leaf, necessitating scans of all training data within the current subtree. Given that tree ensemble algorithms typically comprise over a hundred trees, each with around ten layers, the computational process involves thousands of data passes. This makes the training of tree ensemble algorithms particularly time-consuming for GPUs with datasets with millions of data points and thousands of features.

Hoeffding trees In the domain of decision trees, a limitation of the traditional batch learning methods is their requirement for all data to be held in memory for the learning algorithm to function. This becomes unfeasible when dealing with very large datasets like in the robotics environment, where it is not possible to store all data points in memory simultaneously. To address this, online learning models for decision trees like Hoeffding trees, are used to handle the processing of data sequentially as it becomes available.

Hoeffding trees [8], also known as Very Fast Decision Trees, are an adaptation of decision tree algorithms for online learning scenarios. These trees use incremental learning to make node-splitting decisions based on subsets of data rather than the entire dataset. The methodology includes the utilization of the Hoeffding bound, a statistical measure that calculates the number of samples required to decide a node with a specified level of confidence. Once a decision to split is made and the attribute to split on is chosen, it is considered final and not revisited. Formula 1 shows the Hoeffding bound ϵ where:

- n: Number of samples required.
- R: Range of the variable, i.e. difference between the max and min values it can take.
- δ: Confidence level, i.e. the certainty about the accuracy of our estimate, e.g. confidence level of 95%, means δ=0.05.

$$\epsilon = \sqrt{\frac{R^2 \ln(1/\delta)}{2n}} \tag{1}$$

In the context of online learning with decision trees, like Hoeffding trees, an important factor to consider is concept drift, a concept explained in detail in Section 2.4. Hoeffding Trees, being designed for online learning, encounter concept drift frequently as they continuously adapt to incoming data. These trees must be able to detect shifts in data patterns and adapt their decision-making process accordingly. The effectiveness of a Hoeffding Tree in dealing with concept drift largely determines its performance in real-world scenarios where data distributions are rarely static.

Hoeffding Anytime Tree Hoeffding Anytime Tree (HATT) model [20], also known as Extremely Fast Decision Tree, is an extension of the Hoeffding Tree (HT) framework that aims to address the problem of concept drift by giving the ability to the tree to revise split decisions. More specifically, unlike HT where a split once made is never revisited, HATT evaluates the suitability of existing splits. It works by incorporating a dynamic element that allows for continual adjustment of tree splits. This adjustment is predicated on the evolving nature of data streams, where new data may reveal better-splitting criteria over time. Specifically, HATT can revise a split if further data indicates that the split was not optimal or if a better splitting attribute becomes apparent as more data is accumulated.

One major known problem of decision tree-based models like HATT is their tendency for overfitting, especially when dealing with complex data. Random Forest is an ensemble learning technique that effectively counters this by constructing multiple decision trees during the training phase. Each tree in a Random Forest is built from a random subset of the data and features, which introduces variability and reduces correlation among the trees. This diversity ensures that individual biases or variances of trees are averaged out in the ensemble, leading to a more generalized model.

Adaptive Random Forest The Adaptive Random Forest (ARF) algorithm [11] introduces the benefits of random forests together with Hoeffding Trees. The architecture of ARF involves the utilization of the Hoeffding Tree as its primary base learner to handle streaming data. To enhance its adaptability to evolving data distributions, ARF incorporates online bagging to facilitate the creation of multiple training subsets, to increase diversity among the base learners.

More specifically, ARF implements a drift detection and response system. Each tree within the ensemble has a drift monitoring module, capable of identifying shifts in data patterns, flagged as warnings. These warnings are preliminary indicators of potential concept drift. In response to a warning, ARF initiates the training of background trees, which are developed in parallel to the primary ensemble but remain inactive in influencing the ensemble's immediate predictions. This strategy allows ARF to prepare for significant changes in data distribution without making adjustments that could compromise model stability. Upon confirmation of a drift, a transition from a warning to a recognized change in data distribution, the tree that detected the drift is replaced by its corresponding pre-trained background tree. This method ensures that the ensemble remains updated with relevant changes in underlying data distribution while maintaining stability and avoiding unnecessary adjustments that could lead to

overfitting.

Mondrian Forest A notable limitation of Hoeffding Tree-based models lies in their processing speed. The inherent slowness can be attributed to their reliance on the Hoeffding bound for decision-making at each node, which requires a significant amount of data to achieve the desired confidence level in decision-making. Mondrian Forests uses another approach for split decisions and shows a speed advantage by an order of magnitude while matching accuracy with batch random forest methods trained on identical datasets [17].

Mondrian Forest (MF), developed by Lakshminarayanan *et al.* [17], uses as underlying tree structure the Mondrian process, a stochastic process that generates random partitions of the feature space considering the distribution of the data. Figure 1 shows a visualization of the Mondrian tree, which is similar to decision trees in their splitting mechanism. The main difference with decision trees lies in the consideration of the entire feature space for splitting, while Mondrian trees restrict their splits to the limits defined by the observed data. In other words, if a novel data point falls outside the hyper-rectangle boundary formed by the existing data points within a category, it will not be classified under that category, regardless of its proximity to that boundary. Practically, this helps for the detection and adaptation of drift in the model, since we do not make assumptions about the space that is unknown to us. The only limitation of Mondrian Forests compared to Hoeffding-based tree models is the higher space complexity used by the model [30], which is studied in the next sections.

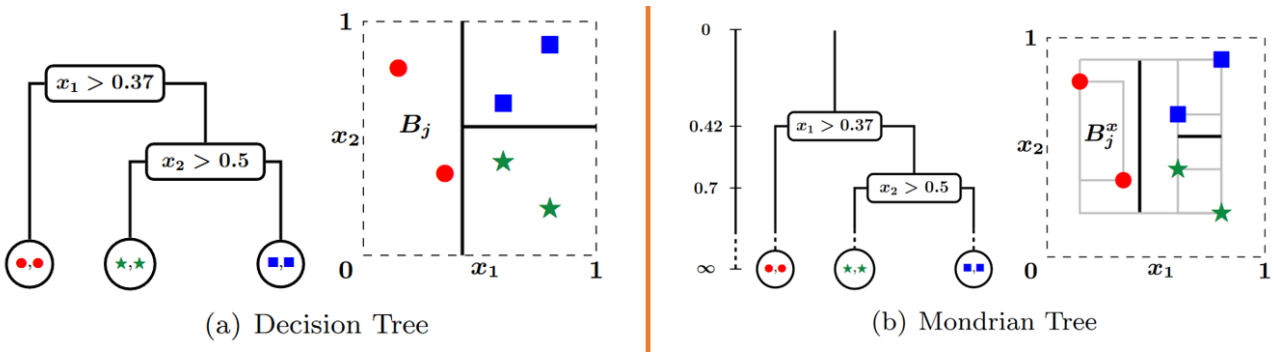


Figure 1: Visual comparison between Decision Tree (left) and Mondrian Tree (right), where x_1 and x_2 are the features. The decision tree segments the entire space based on decision rules (black lines), while the Mondrian tree extends this by creating partitions (gray rectangles) that capture the maximum extent of the existing data for that split. Source [17].

2.3 Model Inference Optimization

In this section, we show the techniques used for improving the performance of decision tree-based models, with a focus on memory footprint and wall time. Wall time, or real-world time, is the actual duration taken by a process or algorithm to complete

a task in a computing environment, including all delays like system resource waits and input/output operations. The goal for many model optimization techniques is to achieve memory and wall time improvements while keeping the decline in accuracy non-existent.

This effort intersects two main areas of research: machine learning, which focuses on refining theoretical models for online learning, and computer systems, which aims to increase the efficiency of implementation and deployment. We explore potential optimizations in both domains that might be relevant to our research. In particular, the investigation looks at decision tree-based models and targets the ARM architecture.

2.3.1 Theoretical model optimization

Hoeffding Trees Memory Optimization Research by Kirkby [15] aims to limit the memory consumption of the Hoeffding tree algorithm. This is achieved by controlling the number of tree nodes and deactivating the least promising leaves. Leaves are evaluated based on their error rates and the probability of examples reaching them. The tree undergoes periodic memory checks, during which memory usage is evaluated and adjusted by deactivating the least promising nodes and reactivating more promising ones to stay within memory limits.

Mondrian Tree Memory Optimization Recent work by Khannouz and Glatard [14] has advanced the online Mondrian forest classification algorithm for use in scenarios with limited memory. The authors introduced five strategies to manage out-of-memory conditions in Mondrian trees, facilitating updates with new data when the memory limit is reached. Among these, the Extend Node strategy is notable for its performance over other methods. It works by continuing to update existing nodes while halting the creation of new ones once the memory limit is reached. Each new data point is still processed and contributes to the updates of the nodes' statistical information and their corresponding dimensional boundaries, i.e. the box of each hyper-rectangle. Additionally, this paper developed mechanisms for node trimming to improve the robustness of Mondrian trees against concept drifts. The trimming process selectively prunes less informative nodes, thus allocating memory to the most relevant parts of the tree, effectively balancing memory utilization against the need for model adaptiveness and precision.

2.3.2 Memory layout optimization

Memory layout is an area of interest in studies to optimize CPU cache usage. This interest arises since accessing the L1 cache is approximately 100 times faster than accessing DDR memory [3]. Cache layouts leverage spatial and sequential locality. Spatial locality refers to the tendency of a process to access items that are near those recently accessed [6]. Sequential locality, a specific type of spatial locality, occurs when data elements are accessed in a linear sequence, such as iterating through elements in a one-dimensional array.

Sequential and spatial locality are beneficial due to the interaction between a computer's cache and the memory from which it loads data. This interaction is based on fixed-size chunks rather than arbitrary-sized requests, which helps streamline organization and minimize overhead. The chunk size depends on the specific hardware. For instance, SRAM caches typically use 64-byte "lines," DRAM employs 2-4 KB "rows," and flash storage or disk drives utilize 4 KB "pages" [5].

The process of loading in chunks allows for the loading of multiple items simultaneously if space permits. For example, as shown in Figure 2, when item 1 of the vector is requested, the block returns items 1 to 8, and these items are loaded if there is enough space available in the cache.

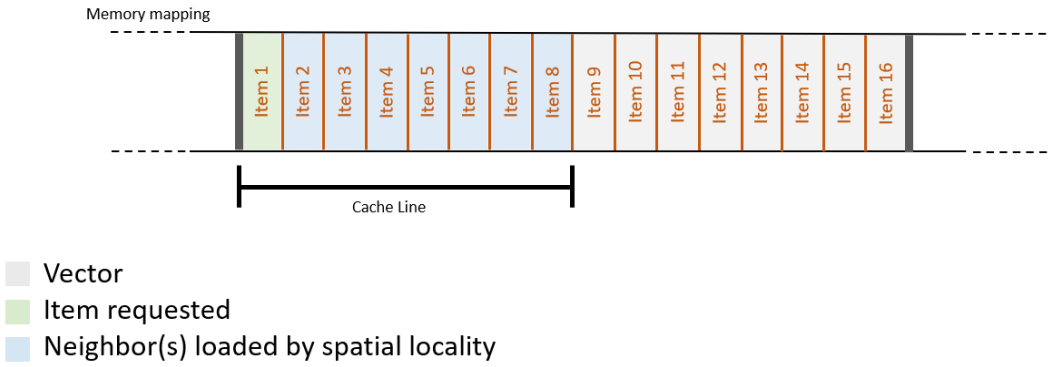


Figure 2: Example of cache loading a subset of a data block. In the example, when item 1 is requested, the cache load thes subset item 1 to item 8.

The study by Chen *et al.* [6] applied memory-locality optimization to the decision tree model. Specifically, the authors optimized the if-else tree structure by using array mapping with a Depth-First Search (DFS) approach, in contrast to the Breadth-First Search (BFS) used in native trees, as shown in Figure 3 . This tree construction method uses the probabilities of path traversal during inference to optimize cache behavior, ensuring the most likely paths are prioritized in memory allocation. This approach results in a reduction of both cache misses and execution time by 70% for ARM architecture and 75% for Intel servers.

Additional content Appendix B.2 explores additional techniques for decision tree optimizations. These techniques include Float to Int encoding and Perfect Binary Tree optimization.

2.3.3 Tree metrics

In this section, we examine the average weighted depth (AWD) metric. This metric is used to determine the sample distribution of the tree.

Average Weighted Depth The Average Weighted Depth (AWD) helps understand the distribution of node usage in the tree. As shown in Figure 4, the average depth is

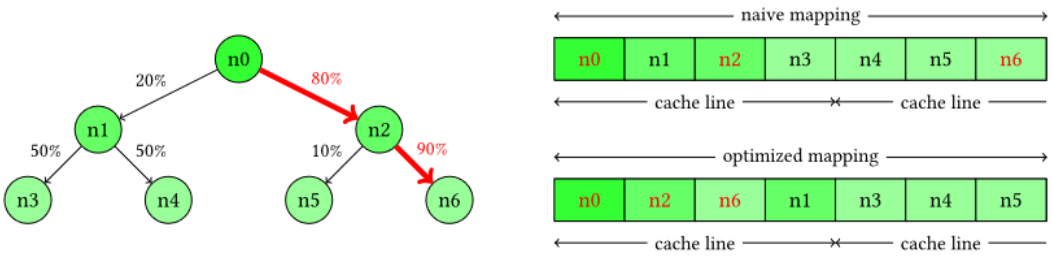


Figure 3: Naive vs. Optimized memory mapping of decision trees. On the left a decision tree with node traversal probabilities. On the top right a naive memory mapping approach where tree nodes are placed consecutively. On the bottom right the optimized mapping, arranging nodes based on the likelihood of access. Source: [6].

calculated by summing the number of leaves for each layer and computing an average. In contrast, AWD enhances this calculation by weighting the depth with the node count at each level. Practically, a high AWD value indicates significant use of the tree's deeper layers, whereas a low value suggests that primarily the upper, shallower layers are used.

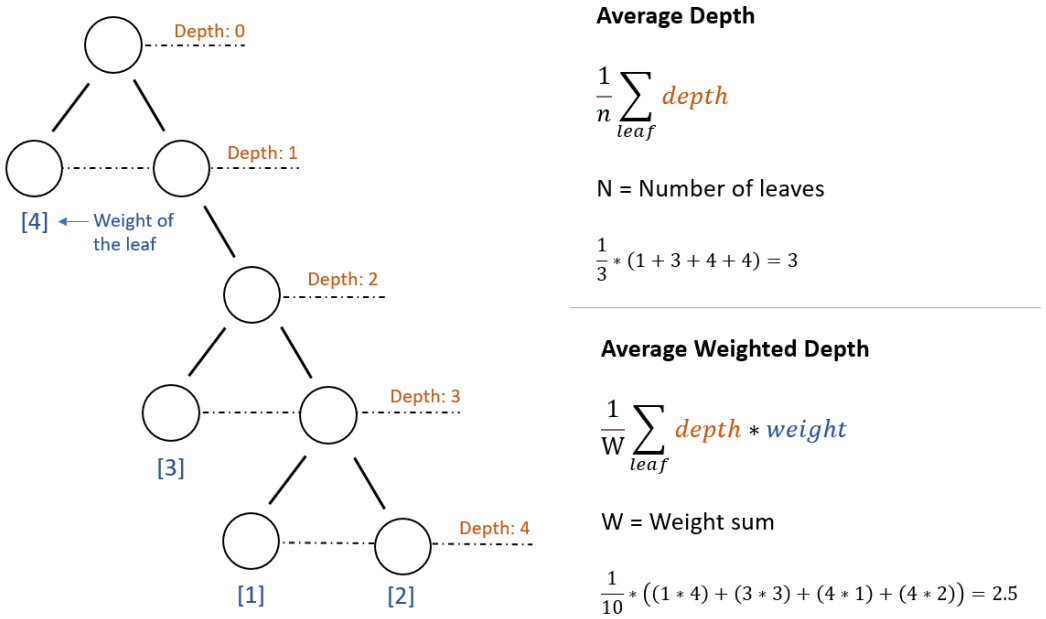


Figure 4: Average depth and average weighted depth metrics compared through an example.

2.4 Drift

One main characteristic in the robotic context to consider while developing a calibration model is how the behavior of the robot evolves over time. This concept is known as drift, and more generically it is defined as changes in the underlying patterns of the

data over time. As explained by Losing *et al.* [19] there are two types of drift: concept drift and virtual drift. Figure 5 visually differentiates between these two types.

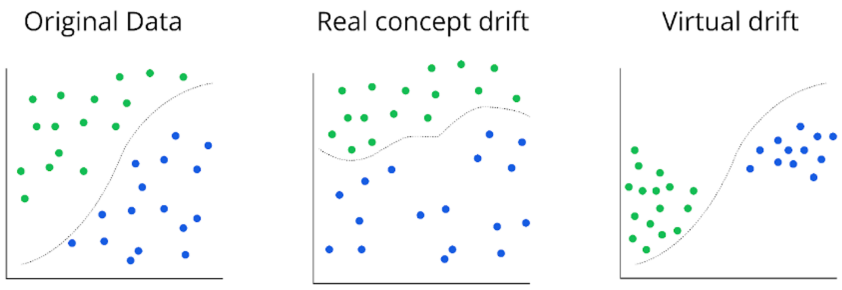


Figure 5: Concept drift (center) compared to Virtual drift (right). Source: [34].

Concept drift Concept drift, also known as real drift, is characterized by changes in the distribution of input data. More specifically, it can be defined as the evolution of data that invalidates the data model. This occurs when the statistical properties of the target variable, which the model aims to predict, undergo unforeseen shifts over time. In a robotic setting, this could manifest as gradual wear and tear of mechanical components or changes in sensor performance. Concept drift can be categorized into types based on the nature of changes in the data distribution. These include mere concept shift, where adjustments to the model parameters can be made smoothly to accommodate the drift, and rapid concept shift, which necessitates the use of active methods.

Virtual drift Virtual drift differs fundamentally from concept drift in that it involves variations in the interpretation or perception of data, rather than changes in the data distribution itself. An intuitive example would be a scenario where a robot repetitively performs the same movement. The data collected in this case would represent only that specific movement, not the robot's full range of capabilities. While such a pattern might initially seem indicative of concept drift, it's actually a case of virtual drift. This occurs because the consistent repetition leads to a narrowed perception of the robot's operational scope, giving the illusion of a drift in the underlying concept when in reality, it's a drift in data perception or collection scope.

3 Methods

In this chapter, we outline the different steps of the thesis project, divided into three phases. The first phase involves model development, focusing on the implementation of a decision model in Rust. The second phase involves algorithmic optimization, improving the model's efficiency. The final phase involves applying this model in a robotic context.

3.1 Model Development

The implementation work of this project aims to implement the Mondrian Forest model in Rust. This implementation is based on different existing Python implementations. Currently, there are two well-known public implementations of the model: River[1] and OneLearn[2]. Both of these implementations are well-established and extensively tested, but they add a lot of complexity as they implement abstract functions to fit multiple models throughout the repository. For this reason, a third implementation is used for guidance. The repository in question is provided by nel215[3]. This repository contains a comprehensible code structure and offers examples that help in the practical application of the algorithm. After the initial porting of the code, adaptation to the River code structure ensureing it follows the same function calls as the River library, thereby improving readability for the River team.

Dataset For the development and testing of the new implementation, we use one synthetic dataset for classification and one for regression, both generated by Scikit Learn. Specifically, we utilize the *datasets.make_classification* and *datasets.make_regression* interfaces. This tool allows us to generate a classification dataset by specifying the number of samples, features, informative dimensions, and clusters per class. For our tests, the number of informative dimensions matches the number of features, thus creating an exponentially more difficult problem as the number of features increases. This methodology is preferred over using standard datasets during development since it provides the flexibility to control the number of samples and complexity, enabling the testing of various aspects including overfitting and underfitting. To ensure reproducibility, the code for generating the datasets used in the experiments is provided in Appendix C.2.

Evaluation The Rust reimplementation is benchmarked against its Python counterpart in terms of execution time, specifically measuring wall-time execution. Wall-time execution refers to the total elapsed time taken for the program to run from start to finish, including factors such as input/output operations and time spent on other processes.

[1] https://riverml.xyz/latest/
[2] https://onelearn.readthedocs.io/en/latest/
[3] https://github.com/nel215/mondrianforest

Performance The performance of the two implementations needs to be comparable, so we monitor it to ensure neither one significantly outperforms the other. For classification tasks, we use accuracy as the performance metric, and for regression tasks, we use mean squared error (MSE).
Accuracy is calculated as:

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}}$$

Mean Squared Error (MSE) is calculated as:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

where n is the number of data points, y_i is the actual value, and $\hat{y}_i$ is the predicted value.

We measure these metrics consistently, as any significant deviation indicates a potential error in the implementation of one or both solutions.

3.2 Model Optimization

In this work, we implement the optimizations described by Chen *et al.* [6]. We focus on implementing array optimization techniques by sorting feature vectors to prioritize sequential access to the most common vector items. This method utilizes spatial locality caching, which, as explained in Section 2.3.2, reduces cache misses and execution time of the program. The comparative analysis involves both the optimized variant and the baseline model developed in the previous iteration.

3.2.1 Tree data structure

In this context, one major part of this phase involves understanding how to best store the features of the model, e.g., in vectors, arrays, or structures, to adapt the code for cache optimization techniques. The common data structure implemented in Python, named tree vector representation, does not take advantage of spatial locality caching. For this reason, the tree struct representation, presented in the work by Tabanelli *et al.* [28], is implemented in our project. Below are the two implementations in comparison.

Vector Representation The memory layout in this data structure consists of multiple vectors, where each vector represents one attribute of the tree, and each item of the vector corresponds to one node. For example, as shown in Listing 1, one vector corresponds to the split value of each node, and another vector represents the split feature.

Struct Representation The second data structure uses a single vector for the entire tree, where each item contains a struct of the node. As shown in Listing 2, the struct contains both the branch feature and the branch split value. When accessing an item of the vector, all the values needed are available, respecting memory locality.

```
Vector<int> left_pnt;
Vector<int> right_pnt;
Vector<float> split_value;
Vector<bool> split_feature;
...
```

Listing 1: Representation of the tree with vectors.

```
Struct Node {
    int left_pnt;
    int right_pnt;
    float split_value;
    bool split_feature;
    ...
}
Vector<Node> tree_nodes;
```

Listing 2: Representation of the tree with struct.

Data Structures Differences As shown in Figure 6, the primary difference between the two data structures lies in their use of spatial locality. In the first structure, spatial locality is utilized to load one attribute for many nodes. In contrast, the second structure loads all attributes for the requested node, along with all attributes from the neighboring nodes. As a result, the vector representation loads too many items into the cache, evicting the items before they can be accessed in sequence. This leads to significantly more data cache misses, causing slower execution. On the other hand, the struct representation maintains better cache locality, minimizing cache misses and resulting in faster execution.

3.2.2 Memory footprint

One metric to consider for the optimizations we apply is the memory consumption per node. Below is the list of attributes taken by each node in a 64-bit architecture for both data structures:

- `parent`, `left`, `right` pointers: unsigned integer, 8 bytes
- `time`, `threshold`: float, 4 bytes
- `feature`: unsigned integer, 8 bytes
- `is_leaf`: bool, 1 byte
- `range_min`, `range_max`: vector of floating points, 4 bytes each, shape `[n_features]`
- `n_labels`: unsigned integer, 8 bytes

Classification specific:

- `sums`, `sq_sums`: vector of vectors of floating points, 4 bytes each, shape `[n_labels, n_features]`
- `counts`: vector of integers, shape `[n_labels]`

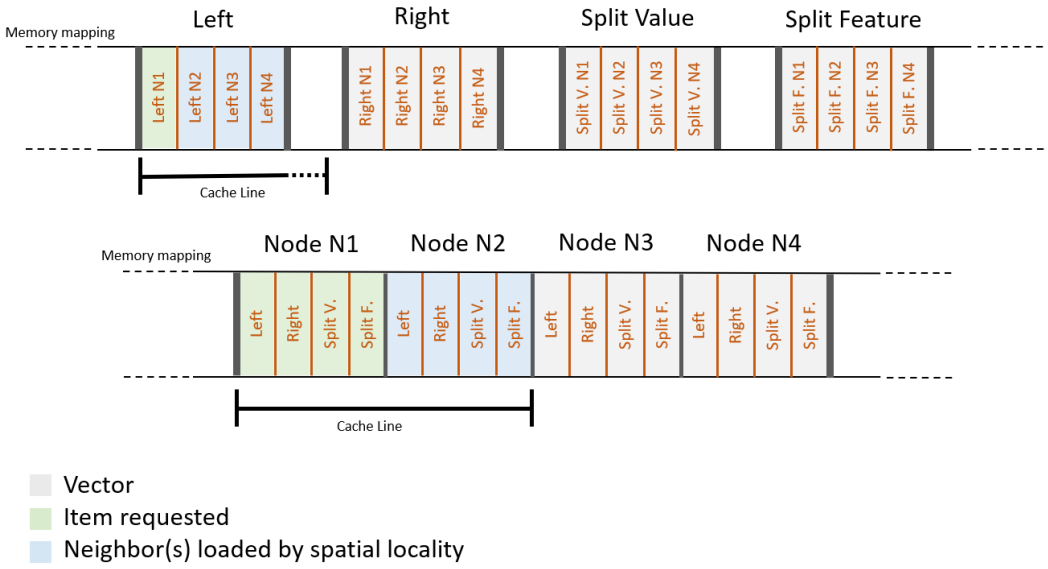


Figure 6: Memory block representation of the Vector data structure (top) and Struct data structure (bottom). The requested item (green) also loads neighboring values (blue). In this example, the cache line can take up to 8 values. For the Vector data structure, it loads all 4 items, while for the Struct, it loads 2 items.

- `n_features`: unsigned integer, 8 bytes

Regression specific:

- `count`: unsigned integer, 8 bytes

Summing up the space taken by each node. The classification task is dependent on the number of labels and features, for a total of $57 + 8n_{\text{features}} + 8n_{\text{labels}} \cdot n_{\text{features}} + 4n_{\text{labels}}$ bytes. Conversely, the regression task is dependent only on the number of features, for a total of $57 + 8n_{\text{features}}$ bytes. Taking an example for classification with 20 features and 5 labels, the size for each element is 557 bytes. For regression with the same number of features, the size for each element is 137 bytes. Considering an average consumer CPU, the amount of L1 cache typically embedded per core is 64 KB. Thus, we can contain 117 nodes in the cache for classification and 478 nodes for regression. Regarding the number of nodes transferred at once from the memory to the CPU, considering paging for DRAM varies depending on the technology, either 2 or 4 KB, we load 3 or 7 nodes for classification and 14 or 29 nodes for regression.

3.2.3 Node access pattern

Mondrian Forest in an online setting differs from the traditional decision tree model as it does not simply traverse nodes from root to leaf. This necessitates further analysis of the node access pattern to understand the impact of the optimizations. In this section, we explore the patterns in different cases of fitting a new record, together with the impact it has on the sorting performance.

Simple Visit When fitting a new record without inserting a new node, the sole action required is updating the node counters. Figure 7 illustrates the steps involved. Initially, in steps A and B, nodes 1 and 2 are visited. Then, in step C, node 3 is visited and its counter is increased by one. Following this, in steps D and E, we backtrack through the stack, incrementing the counters of nodes 2 and 1 by one. Considering the overall impact of this visit, both in the forward and backward steps, we consider neighbor values, thus making use of spatial locality caching.

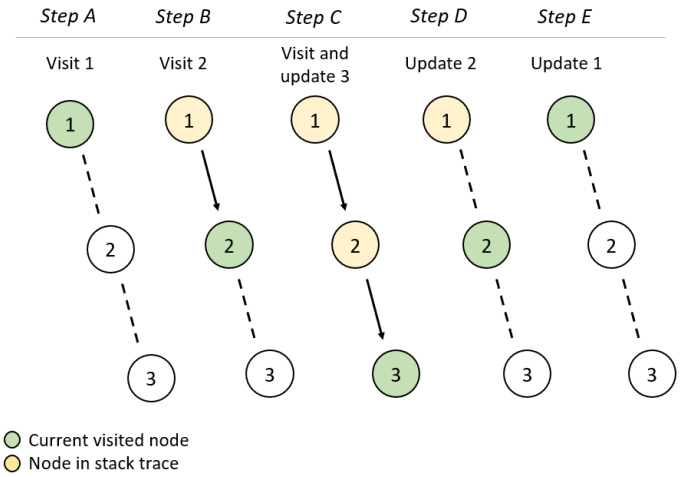


Figure 7: Visited node pattern during a simple visit.

Add a node When fitting a new record and expanding the tree by inserting a new node, the number of nodes we access increases. Figure 8 shows the steps involved when expanding the tree upwards. First, in step A, we visit the child node 2. Here, the algorithm requires that a new node is grown between nodes 1 and 2. Therefore, in step B, we add a new node 5 as the parent of node 2 and node 6 as its sibling. Afterwards, in step C, we go back in the stack and update node 5's counter using the sum of nodes 6 and 2. As a last step, in step D, we update the counter of node 1 using the sum of nodes 5 and 3. In this progression, we see in steps C and D how the succession breaks. In detail, in step D, we update the counter of node 1 with the values from nodes 5 and 3, which means we access three distinct parts of the vector. Generally, in the forward steps until the creation of the new node, we take advantage of spatial locality for caching, while we do not make use of it during the backward steps.

Impact of node access patterns The number of simple visits and visits that require adding a new node are not constant throughout the execution of the algorithm. Online Mondrian forests require the creation of new nodes depending on the additional size of the space we split. The smaller the volume we include in a new split, the less likely we are to split. This translates into having a smaller number of splits later in the execution of the program. If we apply sorting of the nodes to optimize caching at the point where the program creates a negligible number of new nodes, we can still have the benefits of cache optimizations even in the fitting steps of the algorithm.

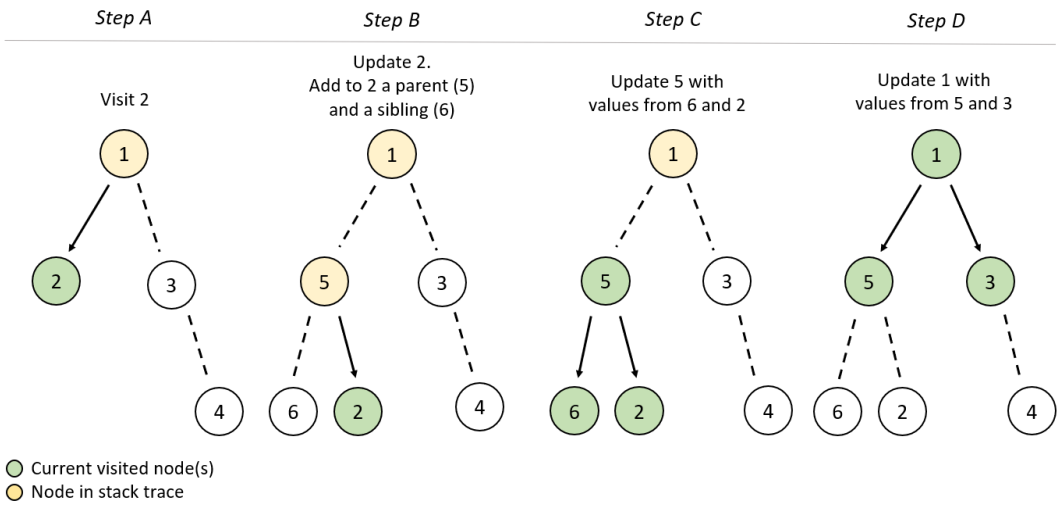


Figure 8: Visited node pattern during insertion of a new node. The new node is inserted as a parent of an existing node.

3.2.4 Evaluation

Evaluation of this method considers the number of L1 cache misses and wall time execution. Additionally, a secondary criterion is the model's accuracy, which we measure to ensure consistency across various versions of the project.

Time Measurement When measuring execution time, we record the wall time. For each iteration, we consider only the time for the fit and predict steps, excluding steps like loading values from memory and sorting the vector.

Removing Variability To accurately measure the impact of caching optimization in this project, certain parameters of the model are fixed. We limit the number of trees to one and set a maximum number of nodes, allocating space for them at the beginning of the execution. This approach prevents the inclusion of the time-consuming process of reallocating the vector when it exceeds its maximum size, which could skew the results. These measures ensure that the results are not affected by such anomalies and eliminate outliers that might otherwise occur in the findings.

Removing stochasticity When measuring time execution and cache misses to measure performance improvements in online machine learning methods, consistency across tests is important. In offline methods, the most widely used strategy, as used by [6], is to separate training and validation, measuring the performance gap only in the latter. Specifically, they fully train the model on the training dataset and then apply the optimizations only in the inference phase, which yields consistent results across runs in decision trees. In online learning, this is not possible since an iteration includes prediction followed by fitting of that same sample. The Mondrian Tree involves a random process during training, which means we generate a different tree for each

run, making fair benchmarking between optimized and non-optimized versions of the model impossible. Specifically, the Mondrian Process involves stochasticity by sampling values from a distribution. To ensure consistent results from sampling, we calculate, instead of sampling, the expected value of that distribution. For instance, instead of using Exponential(λ) we use $1/\lambda$. This does not have any impact on the accuracy of the model if we use only one tree during the execution of the program, and enables consistency between runs.

3.3 Robotic Application

The goal is to understand if we can use the developed model in production and motivate the choices behind it. We test models for regression on one dataset, study the behavior, and discuss techniques needed to run the model on a small compute with techniques regarding model reduction and model limitation.

Data Both versions of the models, regression and classification, are tested on robotics data. The goal is to understand if this model has a low footprint and performs well in terms of accuracy, not only in general tasks, as studied in the previous sections, but specifically in the automation environment. We have one dataset per task, chosen to be large enough to have a measurable impact on performance. Tests show that the regression model processes more than 50,000 samples per second, so datasets smaller than one million samples take a negligible amount of time to execute. However, the implemented model currently processes a single CSV file for execution and must fit in memory. To evaluate the model's effectiveness, we consider both its memory footprint and its accuracy.

Memory Footprint One measurement taken into consideration in this part is the memory footprint of the model. We measure memory footprint using the massif tool by Valgrind. This tool creates snapshots throughout the execution of the program, recording the *Useful Heap Memory*, used for the program's data structures and dynamic allocations, the *Allocator Overhead*, overhead added by the memory allocator, and the total memory usage that sums the two. In the experiments, we are interested in measuring the total allocated memory.

The parameters studied in the memory footprint study are based on the number of trees and the number of features. Moreover, for the classification task, the number of labels is considered.

4 Result

In this chapter, we present the results. All results presented here are obtained on a system running Ubuntu 22.04 within WSL2. The hardware configuration includes an Intel i7-13800H CPU. Each physical core of this CPU has 480 KB of L1d cache, 320 KB of L1i cache, 12.5 MB of L2 cache, and a shared 24 MB L3 cache.

4.1 Model Development

Time Execution We evaluate the Python and Rust models using the synthetic dataset, with reproduction details provided in Appendix C.1. The synthetic dataset generates 2 informative dimensions, 1 cluster per class, 2 features, and 100,000 samples. The test is conducted 40 times to calculate the average and variance. As illustrated in Figure 9, the Rust implementation's execution time for classification with a 95% confidence interval is 2.87 ± 0.01 seconds, compared to Python's 10.48 ± 1.23 seconds. For regression tasks, Rust records an execution time of 0.47 ± 0.01 seconds, while Python takes 13.34 ± 0.18 seconds. This equates to approximately 35,000 samples per second for classification and 213,000 for regression.

(a) Regression **(b)** Classification

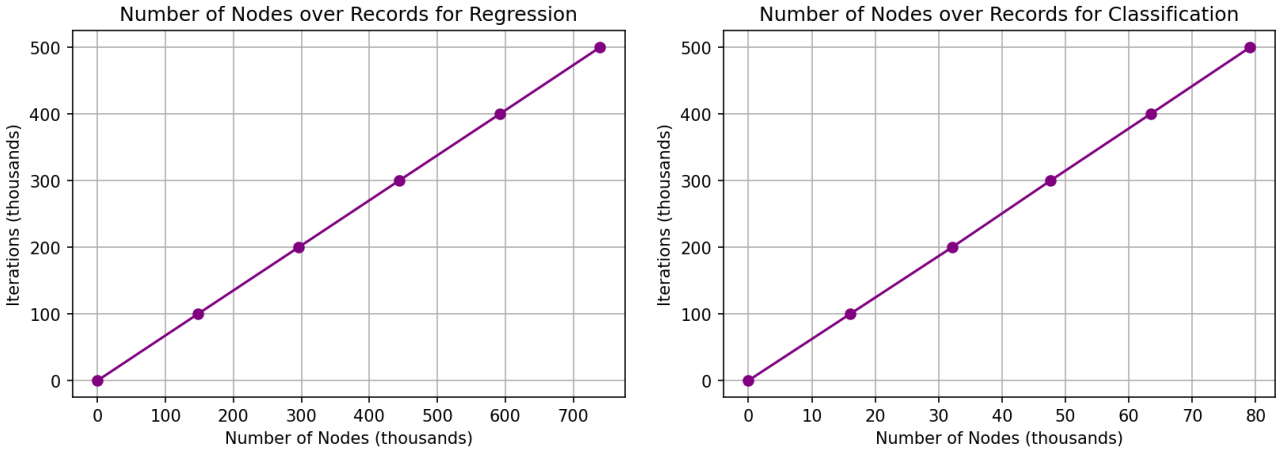
Figure 9: Wall-time execution of Rust compared to Python.

Rust Throughput A common metric in assessing online machine learning models is throughput, measured in terms of records processed and memory usage. We calculate the throughput of the Rust application, assuming that the program inputs CSV files with each float containing 16 digits. For classification, each record includes two features, each requiring 18 bytes, and the label requiring one byte, totaling 37 bytes per record. In regression, we have the same number of features, but the output value is one float, resulting in 54 bytes per record. Based on the average execution times, the throughput for classification in Rust is 1.29 MB/sec, and for regression, it is 11.49 MB/sec.

4.2 Model Optimization

In this section, we examine the results of the model optimization strategy. The details for reproducing these experiments are provided in Appendix B.1.

The results are based on a synthetic dataset designed so that the number of nodes in the decision tree grows linearly, as illustrated in Figure 10. We adopt this strategy to ensure that any correlation between execution time and the number of nodes in the tree can be excluded in subsequent results. Both datasets comprise 500,000 samples. At the end of the training, the classification task tree results in 79,127 nodes, while regression generates a tree of 739,599 nodes. This indicates that the probability of adding a new split, which introduces two new nodes, is approximately 8% for classification and about 74% for regression.



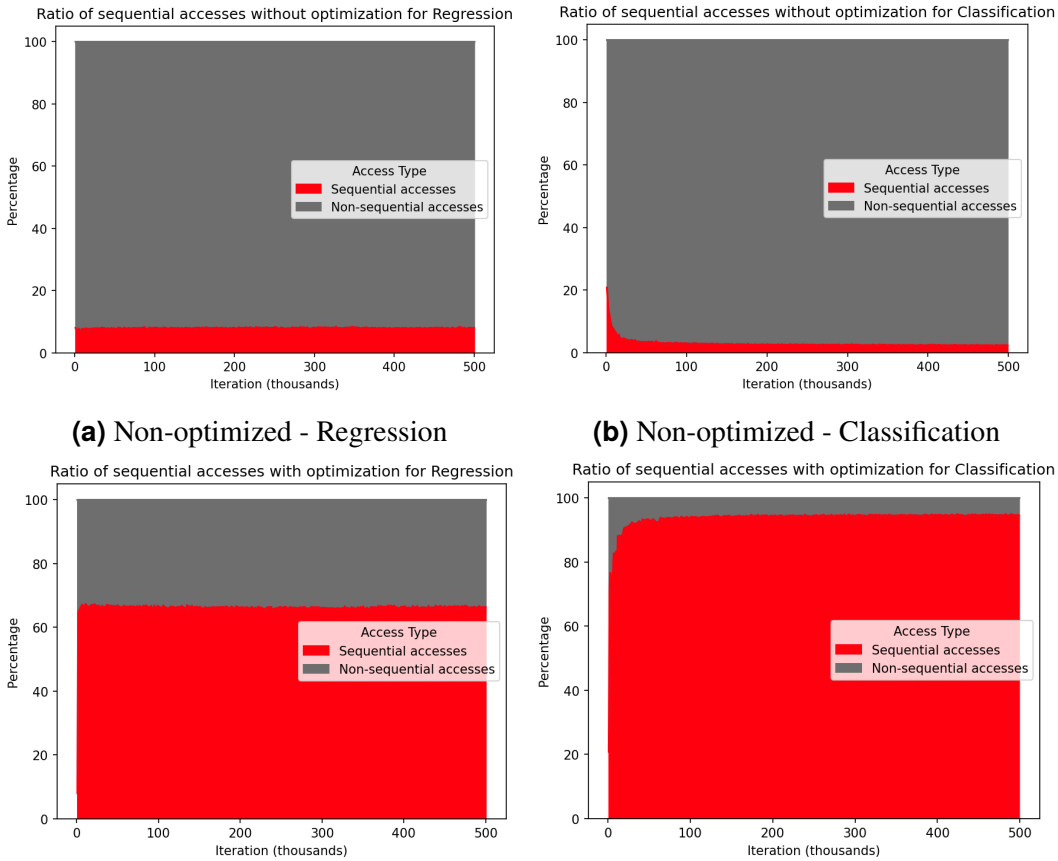
(a) Regression **(b)** Classification

Figure 10: Node count during program execution with the synthetic dataset. Data is measured every 1,000 records.

Sequential Accesses The impact of our algorithm is measured by the number of nodes accessed sequentially during execution. Sequential access is defined as visiting two nodes that are neighbors in the vector during tree traversal. For instance, visiting the node at position 5 in the vector followed by the node at index 6 is sequential, whereas visiting index 5 followed by index 7 is non-sequential. This metric is measured during the inference step for each record processed, from root to leaf. Figure 11 shows that sequential accesses increase from 2% to 94% for classification and from 8% to 66% for regression.

4.2.1 Time execution per iteration

In this section, we present results regarding the execution time for each iteration, which involves two steps: training and inference. To visualize the execution time throughout the program, the data is grouped into chunks of 50,000 iterations. This chunk size is



(a) Non-optimized - Regression

(b) Non-optimized - Classification

(c) Optimized - Regression

(d) Optimized - Classification

Figure 11: Ratio of sequential node accesses during execution.

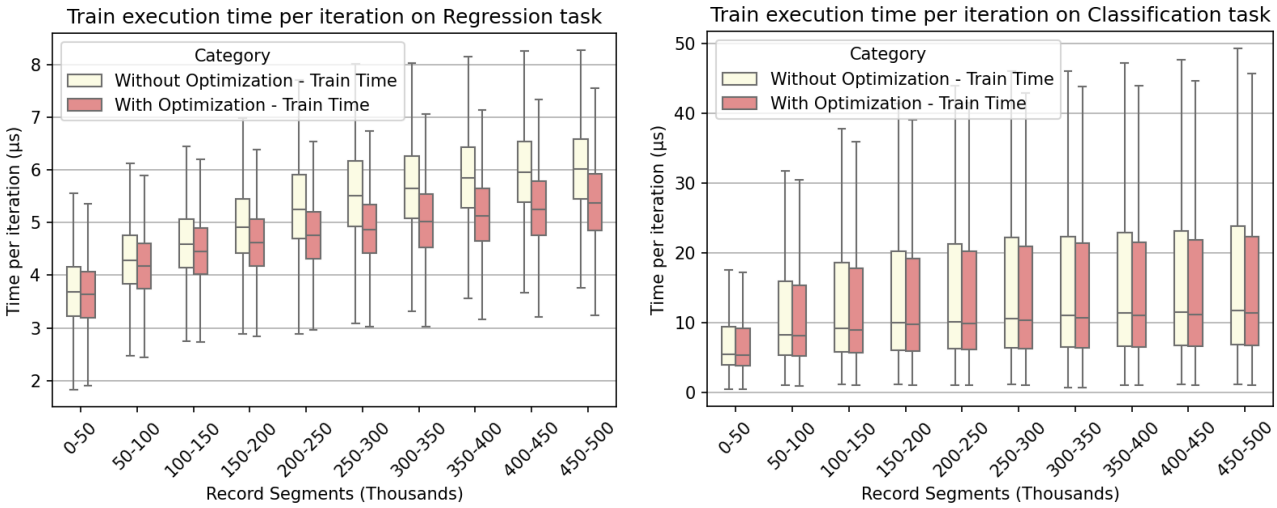
selected to balance having a small window size while avoiding excessive variability between chunks. In these experiments, the algorithm applies optimizations for every 1,000 samples, and the time taken to apply these optimizations is not included in the iteration time. Moreover, the results are not biased by variations in the tree's shape since the process is deterministic; thus, the tree generated in all the experiments is the same, making the median value shown in the plots a highly accurate performance metric.

Optimization overall impact The total execution time for each iteration is shown in Figure 12. The total execution time includes both training and inference steps. Throughout the program's execution, the time taken by both the base model and the optimized version increases. In the last iteration, there is a median improvement of 18.5% for regression and 8.2% for classification with the optimized version. To understand the reason for this performance gap between the two tasks, it is necessary to separate the training and inference steps.

(a) Regression **(b)** Classification

Figure 12: Execution time per iteration, including both training and inference.

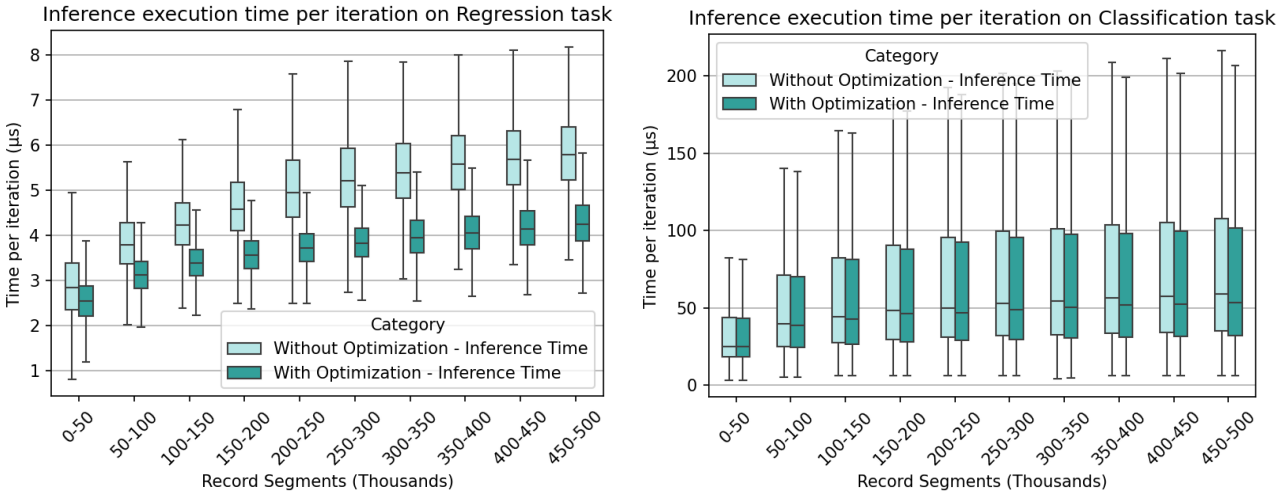
Optimization in Training The impact of our optimization algorithm on the training step is shown in Figure 13. The results indicate a relatively small improvement between the optimized and non-optimized versions of the algorithms, with an improvement of 7.1% for regression and 5.8% for classification.



(a) Regression **(b)** Classification

Figure 13: Impact of optimization on the training step.

Optimization in Inference The optimizations for the inference step demonstrate considerable improvements in performance, with a 34.4% enhancement observed in the regression task and an 11.2% improvement in the classification task.



(a) Regression **(b)** Classification

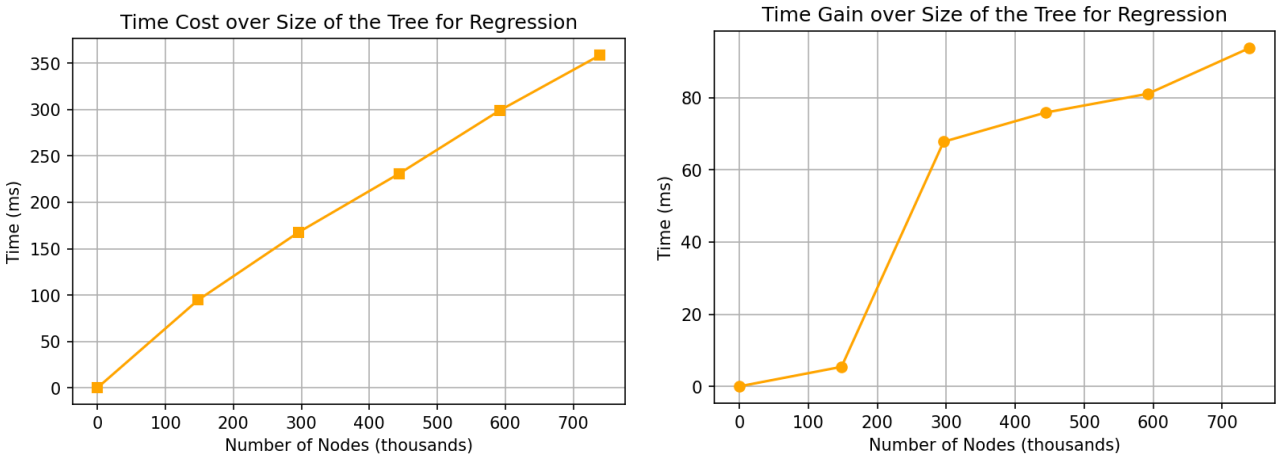
Figure 14: Optimization impact in the inference step.

Optimization Cost and Gain Results shown in Figure 15 present the time required to implement the optimizations and the resulting performance gains relative to the number of nodes in the decision tree. This experiment uses the regression model, with optimizations applied every 100,000 iterations. At each optimization point, the time required to update the model is recorded, and the cumulative performance gain since the last optimization is plotted. The results indicate that at the 400,000th iteration, applying the optimization results in a vector sorting time of 0.298 seconds, with a total time gain of 0.092 seconds over iterations 400,000-499,999. Considering that the regression model adds nodes 74% of the time for our synthetic dataset, and assuming a linear relationship between the number of nodes and both time gain and time cost, we achieve a gain in execution time if nodes are added less frequently than 16% of the iterations.

4.2.2 Motivation behind optimization

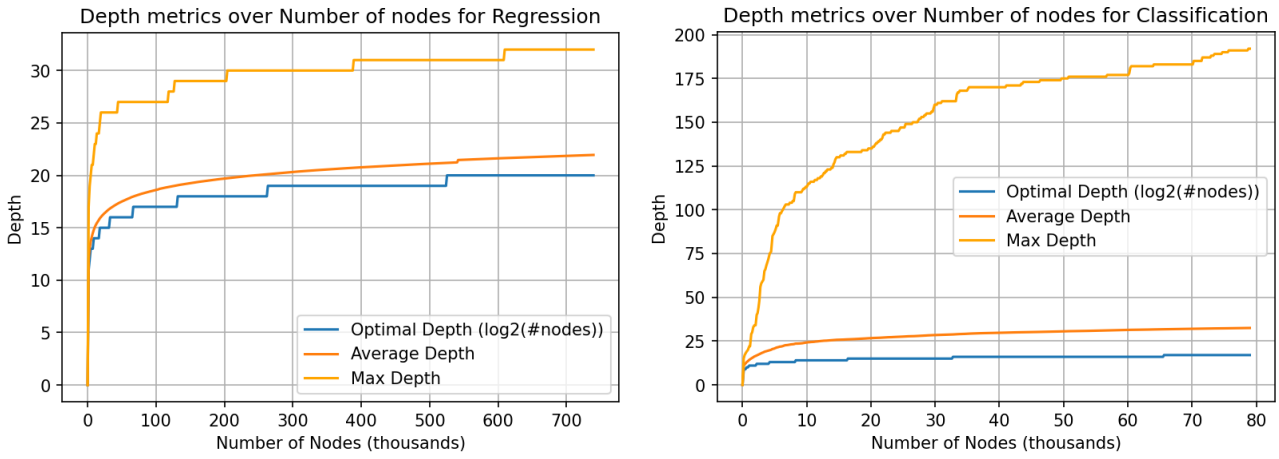
In this section, we present the depth metrics necessary to understand the behavior behind the performance observed in the execution time plots and the performance gap between regression and classification.

Depth Metrics The shape of the tree is analyzed using three depth metrics: maximum depth, average depth, and optimal depth. Maximum depth is the length of the longest path from the root to a leaf. Average depth represents the mean depth of all nodes in the tree. Optimal depth is the theoretical minimum depth for a perfectly balanced binary tree, which is the upper bound of the logarithm base 2 of the number of nodes. As illustrated in Figure 16, the maximum depths are 32 for classification and 192 for regression, the average depths are 22 and 32, respectively, while the optimal depths are 20 and 17, respectively.



(a) Time spent **(b)** Time gained

Figure 15: Sorting time gained vs. spent.



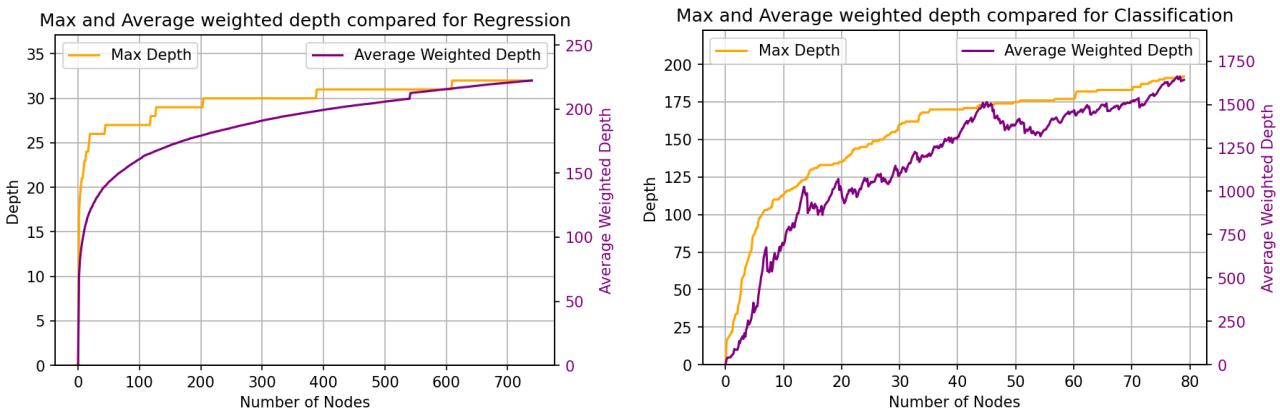
(a) Regression **(b)** Classification

Figure 16: Depth metrics including optimial depth, average depth and max depth.

Average Weighted Depth The following hypothesis examines the potential correlation with average weighted depth (AWD). As shown in Figure 17, the AWD shows an overall upward trend in both instances, following the trend of the maximum depth.

4.3 Robotic Application

This section evaluates the model's memory footprint and performance within a robotic application context. We use one dataset for each task: regression and classification, to assess the model's performance in different predictive scenarios. The experiments run the model across the entire dataset, taking periodic snapshots to measure the memory footprint throughout the program's execution. The memory consumption reported in



(a) Regression **(b)** Classification

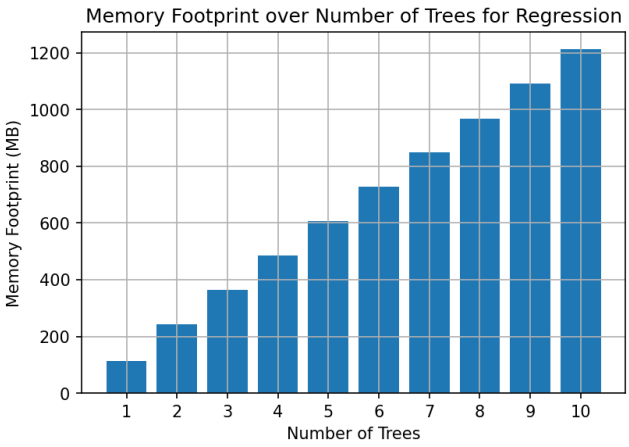
Figure 17: Average weighted depth.

the plots represents the total memory at the snapshot with peak usage. The datasets used in these experiments, along with the data cleaning and feature engineering steps, can be reproduced following the instructions provided in Appendix C.4.

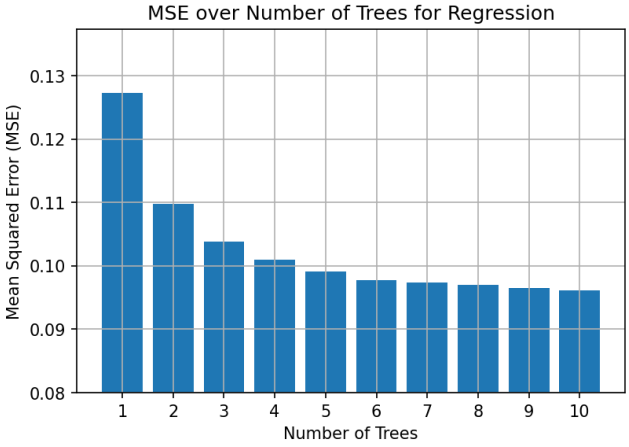
The regression task includes the One Year Industrial Component Degradation dataset[4]. This dataset predicts the degradation of cutting blades in a Vega shrink-wrapper machine over the span of a year. The configuration used during the experiments, detailed in Appendix C.4, comprises 200,000 records and 8 features. Each tree generated has 384,070 ± 140 nodes. For the classification task, we use the Genesis demonstrator dataset[5], which predicts the operational state of the machine. This dataset includes 16,221 samples with 9 labels and 19 features.

Results in Figure 18 address both memory footprint and performance for each task. The analysis indicates that memory consumption increases linearly with the number of trees. Within a 95% confidence interval, each tree consumes 121.3 ± 3.1 MB for classification and 5.6 ± 0.7 MB for regression. Performance, measured in terms of accuracy for classification, is 0.127 and 0.096 for 1 and 10 trees, respectively. For regression, the mean squared error is 0.888 and 0.936, respectively.

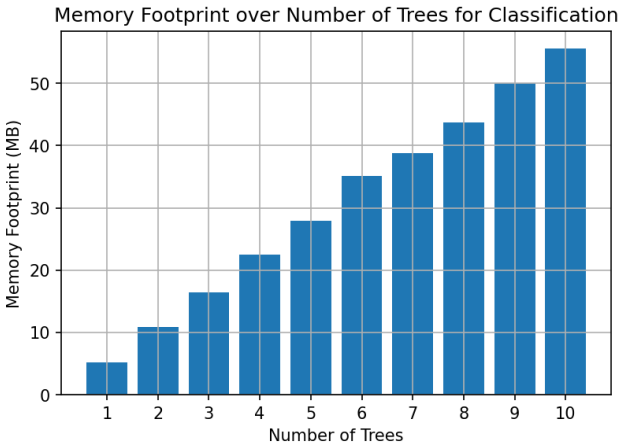
[4] https://www.kaggle.com/datasets/inIT-OWL/one-year-industrial-component-degradation
[5] https://www.kaggle.com/datasets/inIT-OWL/genesis-demonstrator-data-for-machine-learning



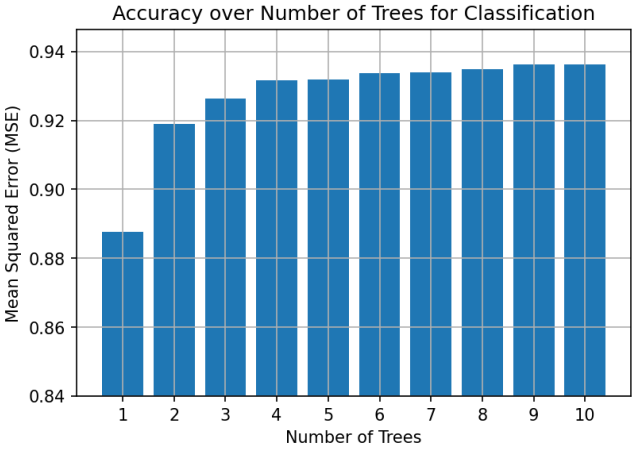
(a) Memory Footprint for Regression



(b) MSE for Regression



(c) Memory Footprint for Classification



(d) Accuracy for Classification

Figure 18: Performance and memory footprint as a function of forest size. Performance is quantified using Mean Squared Error for regression tasks and Accuracy for classification tasks. Forest size is represented by the number of trees.

5 Conclusion

In this chapter, we present the discussion on model development, optimization, and the robotic application of the model.

5.1 Model Development

Time Execution and Throughput The Rust implementation significantly outperforms the Python version, being 3 times faster for classification and 28 times faster for regression. This performance difference can be attributed to the nature of compiled languages like Rust, which generates optimized machine code, in contrast to interpreted languages like Python, which introduces runtime overhead. The throughput results reflect the efficiency of Rust in processing large datasets quickly, particularly in scenarios involving regression tasks. The disparity in speed between regression and classification models is further analyzed in subsequent sections.

These numbers represent the lower bound of performance that we can guarantee. It is important to note that these throughput figures are not fixed and can vary depending on the model's hyperparameters, such as the number of features and labels. Follow-up tests show that the model's speed is not significantly affected by an increase in the number of features. In fact, there is almost a fivefold increase in throughput with five times more labels, indicating that the throughput does not diminish significantly with additional features. This metric is chosen as the guaranteed lower bound for processing speed. Therefore, much better throughput could be achieved with real-world datasets.

5.2 Model Optimization

Sequential Accesses The implementation of the vector sorting algorithm results in a substantial increase in sequential accesses, with improvements of 58% for regression and 92% for classification. This demonstrates that the algorithm provides a significant enhancement in both cases. From the plots, it is evident that the ratio of sequential accesses stabilizes early in the program's execution: around 2,000 iterations for classification and about 30,000 iterations for regression. This suggests that it may not be necessary to sort as frequently as every 1,000 iterations. However, even if sorting is delayed until after the ratio of sequential accesses has converged, any newly appended nodes may still remain unsorted, potentially affecting performance.

Optimization time result To evaluate the impact of vector sorting optimization on the execution time of the Mondrian Forest, we address **RQ1**: *How does vector sorting optimization on the online implementation of Mondrian Forest affect the model's execution time?* The results indicate that the optimization yields an 8% speedup for classification and an 18% speedup for regression. This enhancement in performance can be primarily attributed to improved utilization of spatial locality.

Why is Classification Slower? The disparity in speed between regression and classification can be attributed to several factors:

- The amount of computation required to process one sample in classification is higher, as each node must store and process statistics for all classes present in the dataset. The optimized models indicate that, on average, a full iteration for classification takes 65.1 ms, compared to 9.6 ms for regression, showing a substantial slowdown in the classification model.
- The inference step in the Mondrian Forest for classification involves variance-aware estimation to account for data variability, which requires multiple vector multiplications. This is reflected in the results, where the inference step is, on average, 5 times slower than the training step.
- The classification task generates significantly deeper decision trees compared to regression. In our experiments, the average depth and optimal depth are 20 and 22, respectively, for regression, while they are 17 and 32, respectively, for classification. This means that the model for this task generates a tree that is much more unbalanced, which may affect the performance of the model.

These factors contribute to the following outcomes:

- The computational time significantly outweighs the time saved through spatial locality caching. Although the optimization technique reduces the iteration time by 2 to 5 ms, this saving is minimal compared to the total computation time of around 60 ms per iteration.
- Due to the larger node size in classification, fewer nodes can be loaded into memory with spatial locality. As explained in Section 3.2.2, the average node size for classification is four times greater than for regression. This results in a lower number of nodes stored in the cache and cache lines, necessitating more frequent access to RAM.

Node Access Pattern Analyzing the breakdown of processing times for training and inference in the regression model reveals that both steps benefit from better utilization of spatial locality. Specifically, for the training step, the non-optimized model starts at 6.0 ms and shows an 11% improvement, while the inference step begins at 5.8 ms with a 27% improvement. This indicates that the processing times for training and inference are roughly the same for the non-optimized model. The primary difference in performance gain is due to the node access pattern. As discussed in Section 3.2.3, inference has a much higher rate of sequential accesses compared to training, which results in greater benefits from optimization for the inference step.

Variability The analysis of execution time per iteration reveals significant variability in the results, as shown by the long whiskers in the box plots. This variability arises from the structure of the tree, where the path from root to leaf can vary substantially based on the number of nodes encountered. Specifically, iteration length is directly influenced by the depth of the tree: deeper paths result in longer iteration times, while shallower paths correspond to shorter iterations. Therefore, this high variance is not

due to an insufficient sample size but rather to the inherent differences in path lengths within the tree. The median value is thus used as a representative measure of the tree's average depth.

How Often to Apply the Optimizations? Addressing **RQ2:** *What is the optimal frequency for applying optimizations?* The results show that in our experiment, it is beneficial to sort the nodes if we add nodes in less than 16% of the iterations. In real-world datasets, we expect the number of nodes added after 100,000 iterations to be much below 1%, thus making the gains multiple times higher compared to the costs of applying the optimizations. In a more general automation context, we should optimize the model once the tree is stable and does not grow many more leaves. To give a safe estimate, we benefit from sorting the model when it is adding nodes in less than 1% of the iterations.

Motivation Behind Optimization An important observation is that the performance improvement from the sorting optimization technique is correlated with the tree's structure. Specifically, the speedup is not linked to the number of sequential accesses but rather to depth metrics such as maximum depth and average weighted depth. As explained in Section 2.3.3, this behavior indicates that the leaves in the bottom layers of the tree are actively used. This suggests that the memory layout optimization has the most significant impact when the tree grows and the bottom layers are frequently accessed. The increased average traversal length in such scenarios enhances the benefits of memory locality, thereby improving the algorithm's performance.

Optimization Limitation

- The optimizations we propose are closely tied to the number of features and, for classification tasks, the number of labels in the datasets we are training on. In cases where there is a very large number of features, such as 100, the paging size may only be able to load one node at a time. This negates the benefits of spatial locality caching, rendering our algorithm ineffective. Therefore, when choosing datasets to apply these optimizations, we must consider the number of features, the number of labels, and the paging size, as explained in Section 3.2.2. It is important to determine in advance if the paging can load multiple nodes simultaneously to ensure the effectiveness of the optimizations.
- The optimizations should also consider the CPU load, especially if multiple processes are running concurrently. In scenarios where other processes frequently interrupt, memory and cache could be overwritten often, leading to diminished optimization results. This necessitates evaluating the operational environment to balance the optimization frequency and the overall system performance.

5.3 Robotic Application

With experiments on real-world datasets, the results show that the model works effectively for various robotics tasks. The two implications regard both performance and memory footprint.

Performance The performance in both tasks exhibits a pattern of diminishing returns. In the context of edge computing, this means that we can choose the forest size based on the performance we expect from the model. In our experiments, the models achieve good results with just 6 trees, with minimal improvements in terms of MSE and accuracy beyond this forest size.

Memory Footprint Addressing **RQ3:** *How does the number of trees impact the memory footprint of the model in a robotic controller?* We examine the results in a real-world context. The memory taken by the model increases linearly with the size of the forest. This simplifies deployment options significantly, as we can accurately calculate the memory footprint in the controller. We can deploy the model only if there is sufficient space, and if the controller is expected to undergo a high load, we can trade off some accuracy by loading a model with a smaller forest.

6 Future works

In this chapter, we explore follow-up studies that can complement our work. This chapter presents ideas for future testing regarding model optimization steps and techniques that could be applied to further adapt the model for deployment in a robotic environment.

6.1 Model development

Tradeoff to Maximize Throughput A follow-up study could measure the tradeoff between the number of features and throughput maximization. As explained in Section 4.1, the observed throughput is low because it represents the guaranteed minimum throughput for every dataset. Therefore, by increasing the number of input features, the throughput should increase correspondingly.

6.2 Model Optimization

Multiple Tree Setting In our experiments, we limit the number of trees to one. Future work should involve testing with multiple trees to determine if the sorting optimizations remain effective in this scenario. We hypothesize that even with multiple trees executed on a single core, the cache locality benefits will persist if the execution order follows that shown in Listing 3. The benefits should remain since both the predict and fit functions run from root to leaf within each specific tree, thus continuing to utilize cache-locality advantages.

```
for each sample:
    for each tree:
        tree.predict(sample)
    for each tree:
        tree.fit(sample)
```

Listing 3: Multiple Tree Setting

One necessary algorithmic change in the implementation to support multiple tree settings is to reintroduce stochasticity so that each tree returns different values. As explained in Section 3.2.4, we replace these variables with expected values to achieve consistent results across different runs. Removing the debug statements in the code is the only step necessary to achieve this.

Export/Load Weights In this project, we measure execution time during the program by computing the wall time. This measurement is effective, but it does not account for the total execution time across all processors when libraries use multi-processing, known as CPU time. To measure CPU time, we cannot rely on in-code measurements as we do for wall time, and instead need a utility like the *time* Linux utility. For this utility to function correctly, the entire program execution must

be included in one executable file. Currently, the program applies optimizations within the same executable file, which means the utility would also count the time taken for these steps.

As shown in Figure 19, the solution is to implement the ability to export and load the weights of the model. This approach effectively splits the program execution into two phases. In the first phase, we fit the model for a fixed number of iterations, then either sort or do not sort the nodes in the tree, and export the weights. In the second phase, we load the weights and continue fitting the tree for a fixed number of iterations. The execution time results from the second phase are then benchmarked, allowing us to compare the performance between the optimized and non-optimized versions.

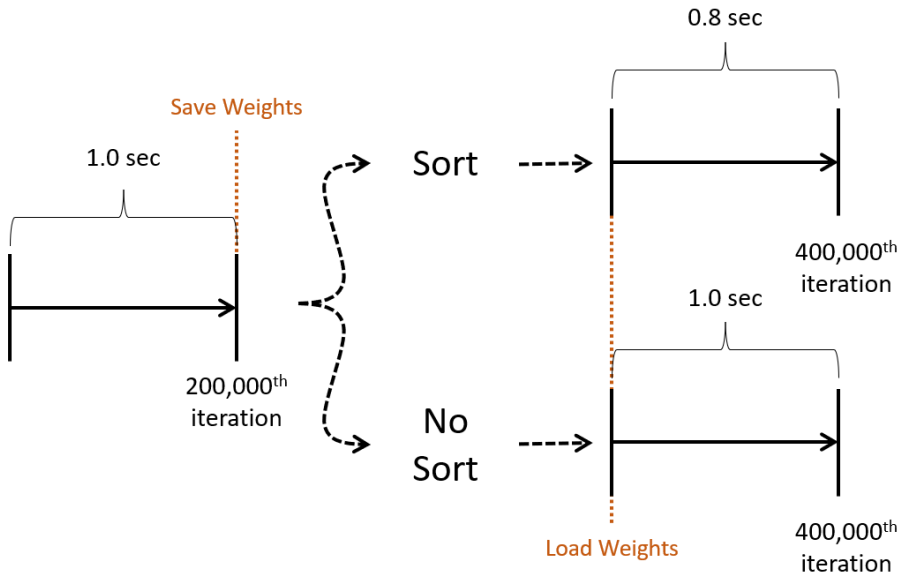


Figure 19: Export and load weights to measure CPU execution time in optimized and non-optimized versions.

6.3 Robotic Application

Support for streaming data In this project we could not process a very large amount of data. Efforts were made to process a large dataset of 3B rows, the Newyork city Taxi Trip Records Dataset[6], but in the process of tranforming Parquet files to CSV, the size of the dataset made in impossible to load the entire dataset in memory. In the future we expect this model to either have the functionality to load partially the file, or streaming the file with a message queuing service such as Apache Kafka.

Base Model In the industrial context, generalization to multiple environments is needed. The optimal scenario involves a model that performs well without prior

[6]https://www.kaggle.com/datasets/microize/nyc-taxi-dataset

experience in the specific environment where it is deployed. The base model would need to be trained offline, with a limitation on the number of layers. We should deliver a base model with fixed parameters, which necessitates implementing a model export and import functionality in Rust.

Adaptive Environment Even with a pretrained base model, decision models like Mondrian Trees do not adapt to new environments as their decisions, once made, remain static during program execution. To address this, models such as Hoeffding Anytime Trees [20] and Adaptive Decision Forests [11] have been proposed, which can revise splits and adapt to new environments. In a robotic context, these adaptive models may yield better results by continuously adjusting to changing conditions.

Model Reduction To further optimize our model, we propose model reduction techniques, which address the challenge of reducing the number of nodes after convergence without losing accuracy. The Multi-Valued Decision Diagrams (MDD) method can be implemented to further optimize the model, improving both memory footprint and execution time.

MDD is a modeling technique developed by Nakahara *et al.* [22] that restructures the decision model to reduce the size in terms of memory. It works by representing multiple decision paths within a single diagram, effectively collapsing redundant nodes and edges to streamline the decision process. As shown in the example in Figure 20, we can represent the decision boundaries more compactly using MDDs compared to binary decision trees. This compact representation reduces both the model complexity and the number of steps required from root to leaf, resulting in a significant performance improvement.

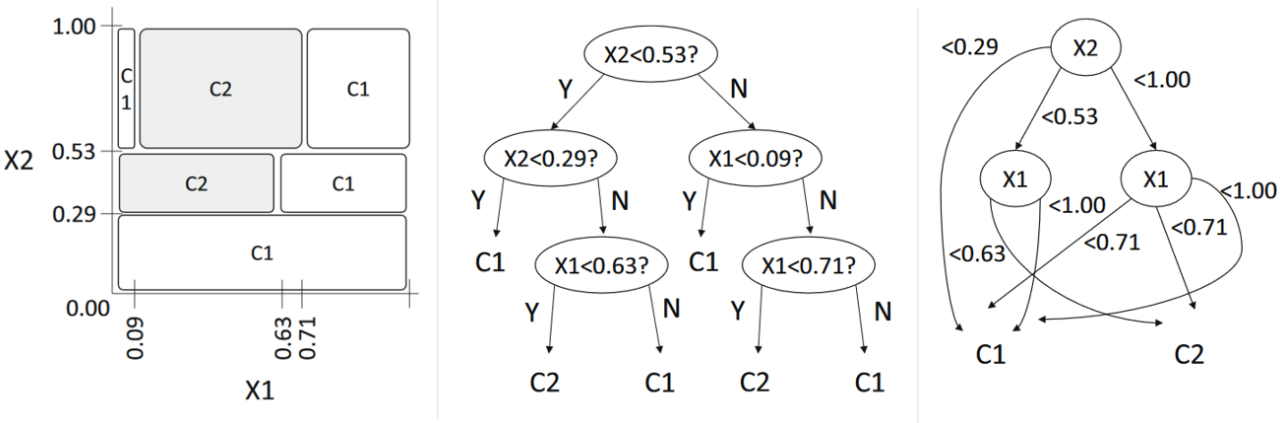


Figure 20: Comparison between BDT (middle) and MDD (right) in a classification space (left). Source [22].

In the Mondrian Forest model used in our study, this approach is not directly applicable in an online learning context but can be suitable for offline methods. Considering the online setting scenario, as shown in Figure 21, during the fitting procedure, the Mondrian Tree model would need a class "unclassified" to store the

volume that does not fall under a specific class. At this point, in order to classify the new record, we would require the information from the hyperbox we fall inside to decide how to split the tree, which we are not storing in the MDD. For this reason, Mondrian forests in an online setting are not implementable. However, we could implement this technique if we eventually switch from online to offline learning. In an automation context, this could happen if we use the online learning algorithm to converge to a certain threshold, transforming the tree into an MDD, and then use that only for inference. This approach could benefit from both the fine-tuning to the environment provided by online learning and the memory savings the MDD provides once we fit the new environment.

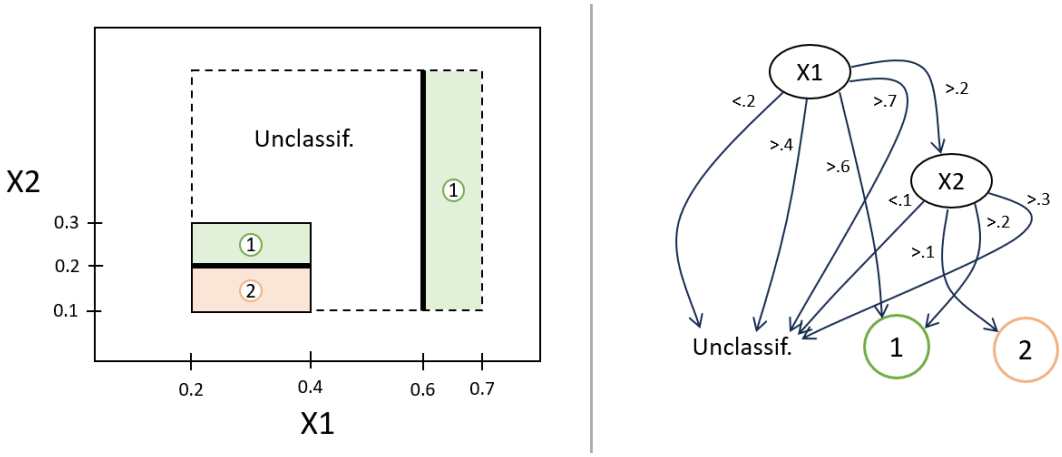


Figure 21: Multi-valued decision diagrams applied to Online Mondrian Tree.

Model Limitation Another approach for reducing the memory footprint of the tree is model limitation, where we limit the growth of the tree. Limiting the depth of a decision tree is an effective way to ensure the model does not grow infinitely. Mondrian Forest already balances the tree well in most cases since it calculates the probability of splitting based on the space it is partitioning, making it less likely to have a split in the deeper layers, especially in real-world datasets. However, there are still datasets that can make the tree grow indefinitely, as shown in our experiments in Figure 10. In our results, the model shown in the plots has been trained on the Synthetic dataset, and in this case, it is not converging given the random sampling characteristics of the dataset. In such cases, limiting the depth of the tree to a fixed height is required so that the robotic compute does not run out of memory.

References

[1] Martín Abadi and Ashish Agarwal. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems, March 2016. arXiv:1603.04467 [cs].

[2] Zouitine Adil, Halford Max, Zouitine Amine, and Di Francesco Marco. LightRiver: Fast and simple online machine learning, 2022.

[3] Andrey. Approximate cost to access various caches and main memory?, November 2010.

[4] Junjie Bai, Fang Lu, and Ke Zhang. ONNX: Open Neural Network Exchange, 2019.

[5] Nathan Beckmann, Phillip B. Gibbons, and Charles McGuffey. Spatial Locality and Granularity Change in Caching, May 2022. arXiv:2205.14543 [cs].

[6] Kuan-Hsun Chen, Chiahui Su, Christian Hakert, Sebastian Buschjäger, Chao-Lin Lee, Jenq-Kuen Lee, Katharina Morik, and Jian-Jia Chen. Efficient Realization of Decision Trees for Real-Time Inference. *ACM Transactions on Embedded Computing Systems*, 21(6):1–26, November 2022.

[7] Katherine Compton and Scott Hauck. Reconfigurable computing: a survey of systems and software. *ACM Computing Surveys*, 34(2):171–210, June 2002.

[8] Pedro Domingos and Geoff Hulten. Mining high-speed data streams. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 71–80, Boston Massachusetts USA, August 2000. ACM.

[9] A.Y. Elatta, Li Pei Gen ., Fan Liang Zhi ., Yu Daoyuan ., and Luo Fei . An Overview of Robot Calibration. *Information Technology Journal*, 3(1):74–78, December 2003.

[10] Stefan Gadringer, Hubert Gattringer, Andreas Müller, and Ronald Naderer. Robot Calibration combining Kinematic Model and Neural Network for enhanced Positioning and Orientation Accuracy. *IFAC-PapersOnLine*, 53(2):8432–8437, 2020.

[11] Heitor M. Gomes, Albert Bifet, Jesse Read, Jean Paul Barddal, Fabrício Enembreck, Bernhard Pfharinger, Geoff Holmes, and Talel Abdessalem. Adaptive random forests for evolving data stream classification. *Machine Learning*, 106(9):1469–1495, October 2017.

[12] Christian Hakert, Kuan-Hsun Chen, and Jian-Jia Chen. FLInt: Exploiting Floating Point Enabled Integer Arithmetic for Efficient Random Forest Inference, 2024. arXiv:2209.04181 [cs].

[13] Zhihong Jiang, Weigang Zhou, Hui Li, Yang Mo, Wencheng Ni, and Qiang Huang. A New Kind of Accurate Calibration Method for Robotic Kinematic Parameters Based on the Extended Kalman and Particle Filter Algorithm. *IEEE Transactions on Industrial Electronics*, 65(4):3337–3345, April 2018. Conference Name: IEEE Transactions on Industrial Electronics.

[14] Martin Khannouz and Tristan Glatard. Mondrian forest for data stream classification under memory constraints. *Data Mining and Knowledge Discovery*, 38(2):569–596, March 2024.

[15] Richard Brendon Kirkby. *Improving Hoeffding Trees*. PhD thesis, The University of Waikato, 2007.

[16] Jens Kober, J. Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32(11):1238–1274, September 2013.

[17] Balaji Lakshminarayanan, Daniel M. Roy, and Yee Whye Teh. Mondrian Forests: Efficient Online Random Forests. arXiv, February 2015. arXiv:1406.2673 [cs, stat].

[18] Zhibin Li, Shuai Li, and Xin Luo. An overview of calibration technology of industrial robots. *IEEE/CAA Journal of Automatica Sinica*, 8(1):23–36, January 2021.

[19] Viktor Losing, Barbara Hammer, and Heiko Wersing. Incremental on-line learning: A review and comparison of state of the art algorithms. *Neurocomputing*, 275:1261–1274, January 2018.

[20] Chaitanya Manapragada, Geoff Webb, and Mahsa Salehi. Extremely Fast Decision Tree, February 2018. arXiv:1802.08780 [cs, stat].

[21] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-Efficient Learning of Deep Networks from Decentralized Data. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*, pages 1273–1282. PMLR, April 2017. ISSN: 2640-3498.

[22] Hiroki Nakahara, Akira Jinguji, Simpei Sato, and Tsutomu Sasao. A Random Forest Using a Multi-valued Decision Diagram on an FPGA. In *2017 IEEE 47th International Symposium on Multiple-Valued Logic (ISMVL)*, pages 266–271, Novi Sad, Serbia, May 2017. IEEE.

[23] Albert Nubiola and Ilian A. Bonev. Absolute calibration of an ABB IRB 1600 robot using a laser tracker. *Robotics and Computer-Integrated Manufacturing*, 29(1):236–245, February 2013.

[24] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Energy efficiency across programming languages: how do energy, time, and memory relate? In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, pages 256–267, Vancouver BC Canada, October 2017. ACM.

[25] Kah Phooi Seng, Paik Jen Lee, and Li Minn Ang. Embedded Intelligence on FPGA: Survey, Applications and Challenges. *Electronics*, 10(8):895, January 2021. Number: 8 Publisher: Multidisciplinary Digital Publishing Institute.

[26] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge Computing: Vision and Challenges. *IEEE Internet of Things Journal*, 3(5):637–646, October 2016. Conference Name: IEEE Internet of Things Journal.

[27] Mahendra Pratap Singh and Manoj Kumar Jain. ISA customization for application specific instruction set processors. In *2015 International Conference on Pervasive Computing (ICPC)*, pages 1–4, January 2015.

[28] Enrico Tabanelli, Giuseppe Tagliavini, and Luca Benini. Optimizing Random Forest-Based Inference on RISC-V MCUs at the Extreme Edge. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(11):4516–4526, November 2022. Conference Name: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.

[29] Stephen Wright and Kerstin Eder. Using Event-B to construct instruction set architectures. *Formal Aspects of Computing*, 23(1):73–89, January 2011.

[30] Haoyin Xu, Jayanta Dey, Sambit Panda, and Joshua T. Vogelstein. Simplest Streaming Trees, October 2023. arXiv:2110.08483 [cs].

[31] Huan Zhang, Si Si, and Cho-Jui Hsieh. GPU-acceleration for Large-scale Tree Boosting, June 2017. arXiv:1706.08359 [cs, stat].

[32] Xingzhou Zhang, Yifan Wang, Sidi Lu, Liangkai Liu, Lanyu Xu, and Weisong Shi. OpenEI: An Open Framework for Edge Intelligence, June 2019. arXiv:1906.01864 [cs, eess].

[33] Fuzhen Zhuang, Zhiyuan Qi, Keyu Duan, Dongbo Xi, Yongchun Zhu, Hengshu Zhu, Hui Xiong, and Qing He. A Comprehensive Survey on Transfer Learning. *Proceedings of the IEEE*, 109(1):43–76, January 2021. Conference Name: Proceedings of the IEEE.

[34] Or Zilberman. Concept Drift Deep Dive: How to Build a Drift-Aware ML System, February 2021.

A Robotic

In this appendix, we explore related works regarding robot calibration techniques.

Why calibrating? One goal of this project is to apply the incremental learning model to improve the robot's calibration model. The calibration model in a robot ensures that the robotic arms' movements align precisely with programmed coordinates, mitigating the risk of errors caused by factors such as vibrational disturbances. The calibration model improves accuracy in robotic arm movements in specific tasks, such as assembly lines, where even minor deviations can lead to significant product defects.

Robot position measurement Calibration methods primarily focus on pinpointing the position of the robot's end-effector, the component at the tip of a robotic arm utilized for tasks like spraying, welding, and handling. One common tool used for measuring the robot's end-effector is the laser tracking system, as shown in Figure A1. This system operates by projecting a laser beam onto a target, typically a sphere fixed to the end-effector, then calculating the end-effector's position in 3D space based on the time taken for the laser light to reflect back. An example of a laser tracking system is by Jiang *et al.* [13], conduct a study using the Leica tracking system. This system, depending on its specific model, exhibits an uncertainty range between 10-300 μm and is capable of collecting data at a rate of up to 1,000 points per second.

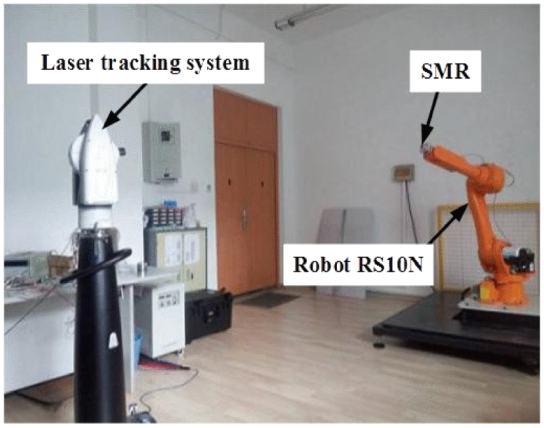


Figure A1: Experimental Setup. Laser tracking system (left), Robot (right). The laser tracking system captures the position of the robot's end-effector. Source: [13].

A.1 Error sources

End-effector error types When we talk about robot calibration, we need to consider the different types of errors that we try to compensate for with the calibration model. As mentioned by Gadringer *et al.* [10], the two distinct errors that should be considered in this regard are positioning error and orientation error:

- Positioning Error: Refers to the difference between the intended or programmed position of the robot's end-effector and its actual position.
- Orientation Error: Refers to the difference between the intended or programmed orientation (angle) of the robot's end-effector and its actual orientation.

Error sources When building a calibration model, we should take into account different error sources. Li *et al.* [18] divides these sources of errors into three major categories:

- Deterministic Error: Remains constant over time and can be measured in advance, e.g., geometric errors in joint connections.
- Time-Varying Error: Changes over time according to predictable patterns, e.g., errors induced by temperature fluctuations in the machine.
- Random Error: Non-predictable and can't be measured beforehand and typically fixed with statistical methods, e.g., external vibration or operational errors.

Each error source brings different challenges that require different models to address them. In the next section, we examine state-of-the-art models that aim to fix deterministic errors.

A.2 Calibration method approaches

In this section, we explore the different existing calibration models to adjust for calibration errors. According to Eletta *et al.* [9], calibration models can be classified into model-based and model-free approaches.

Model-based calibration This method involves creating a detailed mathematical model of the robot, which includes parameters like joint angles, link lengths, and other geometric factors. It may also incorporate non-geometric factors like joint stiffness. The model predicts the robot's behavior and is refined based on calibration data to improve accuracy.

Model-free calibration This approach, also known as non-parametric calibration, doesn't rely on a detailed model of the robot. Instead, it focuses directly on the measurement data from the robot's performance to make necessary adjustments. This approach may use methods like polynomial approximations to compensate for errors directly based on observed data, without needing to consider the underlying physical or geometric model of the robot.

A.3 Calibration models

Various model-based calibration frameworks exist. In this section, we examine the two predominant models used in the literature: kinematic and dynamic.

Kinematic model Serial robots are composed of a series of linked segments connected by joints. The kinematic model describes the relationship between the joints by applying geometry to study the movement. One example of a kinematic model is the D-H model, which according to Li *et al.* [18], is the most used kinematic model as of 2021. It works by fixing the link coordinate system at the link joint, allowing for a simplified representation of joint movements and link orientations.

Dynamics model In practical applications, robotic operations are influenced by forces and torques, diverging from the kinematic model's assumption of rigidity and its exclusion of flexibility or external forces. The dynamics model, which builds upon the foundational kinematic model, incorporates additional layers of physical interactions. These interactions include not only the forces and torques but also account for factors such as inertia, friction, and interrelations among the robot's components. One example of a dynamics model is the stiffness model. As explained by Nubiola and Bonev [23], this model is designed to characterize the elasticity of a robot. This model includes the behavior of each joint as a variable, specifically omitting joints that are oriented along the gravitational axis, thus focusing on those components most affected by elastic deformations under operational loads.

B Extra model optimization

In this appendix, we explore additional model optimizations.

B.1 Model optimization

In this section, we detail how to reproduce the results for model optimization. These steps require the generation of the dataset. The version used to generate these results is commit *5196962*[7]. Below are the instructions to run the four cases, which include classification/regression and optimized/non-optimized scenarios.

```
# Classification - Not optimized
# Set line 55 in 'examples/classification/synthetic.rs' to:
#     const CACHE_SORT: bool = false;
RUSTFLAGS=-Awarnings cargo run --release --example synthetic

# Classification - Optimized
# Set line 55 in 'examples/classification/synthetic.rs' to:
#     const CACHE_SORT: bool = true;
RUSTFLAGS=-Awarnings cargo run --release --example synthetic

# Regression - Not optimized
# Set line 55 in 'examples/regression/synthetic_regression.rs' to:
#     const CACHE_SORT: bool = false;
RUSTFLAGS=-Awarnings cargo run --release --example synthetic-
    regression

# Regression - Optimized
# Set line 55 in 'examples/regression/synthetic_regression.rs' to:
#     const CACHE_SORT: bool = true;
RUSTFLAGS=-Awarnings cargo run --release --example synthetic-
    regression
```

Listing 4: Script Execution for Synthetic Model Testing

B.2 Model optimization

Float to Int encoding The recent work by Hakert *et al.* [12] introduces FLInt, an operator that eliminates the need for floating-point hardware by performing floating-point comparisons using only integer and logic operations. The researchers provide a formal proof demonstrating its ability to preserve model accuracy while eliminating floating-point computations during inference. The implementation of FLInt in low-level realizations of random forests maintains model accuracy, and experimental evaluations on ARMv8 architectures show execution time reductions of approximately 30%.

Perfect Binary Tree One modification we can introduce is the redistribution of the tree nodes. Typically, the tree in this model tends to grow asymmetrically, meaning

[7] https://github.com/MarcoDiFrancesco/light-river-cache/tree/5196962

it develops more branches on one side than on another. Our approach precludes adding more nodes at runtime; instead, we establish a perfect binary tree by allocating the necessary memory at the start of execution and fixing the tree size based on the maximum height it can reach. For instance, a height of 10 would yield 1023 nodes. This strategy imposes a limitation on the original algorithm since it restricts the maximum depth that can be achieved. Furthermore, given that trees in real-world datasets often grow non-symmetrically, this results in a larger memory footprint compared to traditional Mondrian Forests.

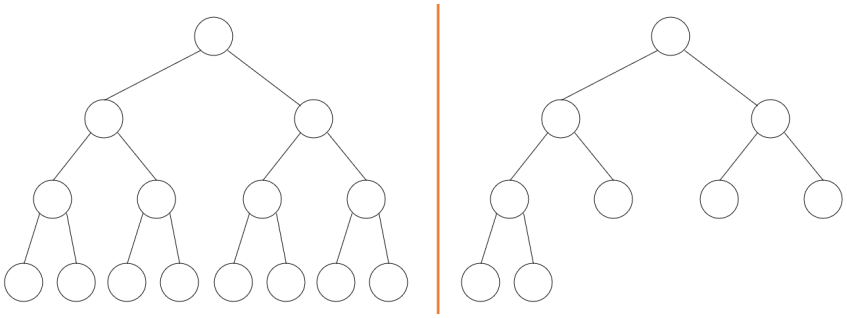


Figure B1: Two decision trees, one binary perfect (left) and one asymmetric (right).

C Reproducibility

In this appendix, we explore the repository structure and commands in detail for reproducibility. This appendix contains only the code that is not already in the repository.

C.1 Model development

Time execution The script runs for 40 iterations. A flag is set for the program execution to avoid showing warnings during the testing of the model. This does not change the performance of the model execution and only makes the script more readable. The flag release is set to avoid memory safety features running in the background. We noticed a 45x speed slowdown during tests if we do not set this flag. In the repository, this command is run at commit *ec2109a*[8]. In the Python script, the prints for accuracy and count nodes were removed, keeping only the one to print the execution time.

```
# Classification
for i in {1..40}; do
  echo "RUN: ${i}" >> run_synthetic_clf_rust.txt
  RUSTFLAGS=-Awarnings cargo run --release --example synthetic >>
   run_synthetic_clf_rust.txt

  echo "RUN: ${i}" >> run_synthetic_clf_python.txt
  python python_baseline_synthetic_clf.py >>
   run_synthetic_clf_python.txt
done

# Regression
for i in {1..40}; do
  echo "RUN: ${i}" >> run_synthetic_reg_rust.txt
  RUSTFLAGS=-Awarnings cargo run --release --example synthetic-
   regression >> run_synthetic_reg_rust.txt

  echo "RUN: ${i}" >> run_synthetic_reg_python.txt
  python python_baseline_synthetic_reg.py >>
   run_synthetic_reg_python.txt
done
```

Listing 5: Script Execution for Synthetic Model Testing.

C.2 Dataset generation

Here are the two variants for generating the classification and regression datasets. In both classification and regression, the number of features and informative features is the same. The number of redundant features is fixed to zero, and the clusters per class are set to one for all the tests conducted for this project. The reason we keep the dataset simple is that the goal of this project is not to measure accuracy extensively

[8] https://github.com/online-ml/light-river/tree/ec2109a

but rather to evaluate performance. Generating more complex datasets would only result in longer wait times during experimentation. Although we also test a higher number of parameters, we found a reasonable waiting time with these parameters.

```
from sklearn.datasets import make_classification
from sklearn.datasets import make_regression
import pandas as pd

# Comment either Classification of Regression
# Classification
n_features = 2
X, y = make_classification(
    n_samples=100000,
    n_features=n_features,
    n_informative=n_features,
    n_redundant=0,
    n_clusters_per_class=1,
    n_classes=3,
)

# Regression
X, y = make_regression(
    n_samples=100000,
    n_features=n_features,
    n_informative=n_features,
)

# Create DataFrame
df = pd.DataFrame(X, columns=[f"feature_{i}" for i in range(1,
   n_features+1)])
df["label"] = y

# Classification
df.to_csv("syntetic_dataset_v?.csv", index=False)
# Regression
df.to_csv("syntetic_reg_dataset_v?.csv", index=False)
```

Listing 6: Dataset generation for classification and regression of the synthetic dataset.

C.3 Valgrind

Valgrind is a programming tool used for memory debugging, memory leak detection, and profiling. The following command runs Valgrind with the Massif tool to analyze the heap memory usage of the program. The output is saved to a file, which can later be interpreted using the `ms_print` utility.

```
valgrind --tool=massif --massif-out-file=massif.out target/release/
   examples/machine_degradations ms_print massif.out
```

Listing 7: Valgrind command for heap analysis.

The results of the heap memory analysis look like this:

```
--------------------------------------------------------------------
```

```
  n         time(i)          total(B)   useful-heap(B) extra-heap(B)
--------------------------------------------------------------------
 77 17,234,778,062      126,234,488      117,305,727     8,928,761
 78 17,395,228,840      126,529,408      117,512,171     9,017,237
 79 17,555,679,528      126,799,728      117,701,395     9,098,333
 80 17,716,131,842      127,093,568      117,907,083     9,186,485
 81 17,764,227,659      127,193,648      117,977,139     9,216,509
```

Listing 8: Result of the heap analysis.

C.4 Robotics Dataset

Regression For the regression test, we merge multiple CSV files containing data on industrial component degradation, preprocess the data by removing the timestamp column, and limit the dataset to 100,000 rows. We then train a RandomForestRegressor with one tree on a subset of 10,000 rows to predict motor torque. The mean squared error (MSE) for this model is 0.0704.

```
import pandas as pd
import os
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error, r2_score

csv_dir = 'one-year-industrial-component-degradation'
csv_files = [f for f in os.listdir(csv_dir) if f.endswith('_mode1.
    csv')]
dataframes = []
for file in csv_files:
    file_path = os.path.join(csv_dir, file)
    df = pd.read_csv(file_path)
    dataframes.append(df)

merged_df = pd.concat(dataframes, ignore_index=True)
merged_df = merged_df.drop("timestamp", axis=1)
merged_df = merged_df.iloc[:100000]
merged_df.to_csv('one-year-industrial-component-degradation.csv',
    index=False)
X = merged_df.drop(columns=['pCut::Motor_Torque'])[:10000]
y = merged_df['pCut::Motor_Torque'][:10000]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
    =0.2, random_state=42)
regressor = RandomForestRegressor(random_state=42, n_estimators=1)
regressor.fit(X_train, y_train)
y_pred = regressor.predict(X_test)
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
print(mse, r2)
```

Listing 9: Dataset generation for regression.

Classification For the classification test, we use a dataset containing state machine labels. We preprocess the data by separating features from labels and split it into training and testing sets. We train a simple RandomForestClassifier with one tree and achieve an accuracy of 0.9238 on the test set.

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, classification_report

state_machine_labels = pd.read_csv('data/Genesis_StateMachineLabel.
    csv')
X = state_machine_labels.drop(columns=['Label'])
y = state_machine_labels['Label']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
    =0.2, random_state=42)
rf_classifier = RandomForestClassifier(random_state=42)
rf_classifier.fit(X_train, y_train)
y_pred = rf_classifier.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
classification_rep = classification_report(y_test, y_pred)
print("Accuracy:", accuracy)
```

Listing 10: Dataset generation for classification.