FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

# Data stream mining on the edge

**João Luís Cardoso Rodrigo**


Mestrado em Engenharia Informática e Computação

Supervisor: Prof. João Pedro Mendes Moreira

Second Supervisor: Profª. Maria Goreti Carvalho Marreiros

August 2, 2024

# Data stream mining on the edge

**João Luís Cardoso Rodrigo**

Mestrado em Engenharia Informática e Computação

August 2, 2024

# Abstract

This thesis focuses on data processing at the edge of a system, specifically through data stream mining techniques with an emphasis on Online Decision Trees. The primary objective is to survey online decision trees and detail the data processing workflow applicable to edge computing.

To achieve this, we investigate the adaptation of decision trees for real-time processing and the challenges involved. We propose optimizations to overcome these challenges and validate our approach through experiments in various scenarios. Our research methodology provides a comprehensive discussion on these topics, concentrating on Edge Devices or Nodes for processing. Among the decision trees, Hoeffding Trees are examined in depth. By employing incremental learning, concept drift detection, improved splitting capabilities, and ensemble techniques, we demonstrate that online decision trees are well-suited for data stream mining.

Our experiments utilized different frameworks and both synthetic and real datasets. Synthetic datasets were designed with custom concept drifts to evaluate the adaptability of various decision trees. The results highlight the effectiveness of Hoeffding Adaptive Trees in maintaining high performance and quick recovery during concept drifts compared to other models.

This thesis offers a detailed overview of Online Decision Trees, particularly Hoeffding Tree implementations, and their application in resource-constrained edge environments. Our findings contribute to the broader field of data stream mining by enhancing the understanding and capabilities of Online Decision Trees.

# Resumo

Esta tese investiga o processamento de dados na periferia de sistemas (Edge Devices), com foco em técnicas de Data Stream Mining, particularmente Online Decision Trees (ODTs). O objetivo principal é analisar as ODTs em profundidade e delinear um fluxo de trabalho de processamento de dados adequado à computação na periferia.

Para atingir este objetivo, exploramos a adaptação de Decision Trees para processamento em tempo real, abordando os desafios inerentes. Propomos otimizações para superar tais desafios e validamos nossa abordagem através de experiências em diversos cenários, utilizando dispositivos de periferia. Entre as ODTs, as Hoeffding Trees (HT) são examinadas em detalhe. Através da aplicação de Incremental Learning, Concept Drift detection, Split enhancements e Ensembles, demonstramos a adaptação das ODTs para Data Stream Mining.

As experiências utilizaram diferentes frameworks e conjuntos de dados sintéticos e reais. Os dados sintéticos foram concebidos com concept-drift específicos para avaliar a adaptabilidade das ODTs. Os resultados evidenciam a eficácia das Hoeffding Adaptive Tree (HAT) em manter um desempenho elevado e uma recuperação rápida durante desvios de conceito, comparativamente a outros modelos.

Esta tese oferece uma análise detalhada das ODTs, com ênfase nas implementações da HT, e da sua aplicação em ambientes de periferia com recursos limitados. As nossas descobertas contribuem para o campo de Data Stream Mining, aprofundando a compreensão e as capacidades das ODTs.

# Agradecimentos

I wish to extend my heartfelt gratitude to the GECAD team for their unwavering support and invaluable guidance throughout my research journey. Special thanks are due to my esteemed colleagues in the Industry 4.0 sector, particularly Francisca, Lara, and Diogo. Your collaboration and insightful discussions have been instrumental in shaping the direction and quality of my work.

I am deeply indebted to my colleague, Afonso Lourenço, for his exceptional assistance. His tutelage on research methodologies and workflow has been a cornerstone of my academic development. Afonso's patience and willingness to help me grasp complex concepts have been invaluable, and I am sincerely grateful for his mentorship and friendship.

I extend my profound appreciation to Professor Goreti Marreiros, whose inspiration and steadfast support have been a driving force throughout this process. Her guidance and encouragement have been pivotal to my achievements, and I am honored to have had the opportunity to work under her supervision.

I would like to express my deepest gratitude to my supervisor, Professor João Moreira, for his remarkable guidance and unwavering support. Stepping in as my supervisor in the middle of the first semester, Professor Moreira provided a seamless transition and continuously guided me with insightful suggestions and constructive feedback. His willingness to connect me with other students for collaborative idea generation greatly enriched my experimental approaches. His dedication and mentorship have been invaluable, and I am profoundly thankful for his contributions to my academic journey.

To all those mentioned and the many others who have contributed to this work in ways both large and small, I offer my deepest thanks. Your support has made this thesis possible, and I am forever grateful.

João Rodrigo

*"You should be glad that bridge fell down.*
*I was planning to build thirteen more to that same design"*

Isambard Kingdom Brunel

# Contents

# List of Figures

# List of Tables

# Abreviaturas e Símbolos

| | |
|---|---|
| IoT | Internet of Things |
| ML | Machine Learning |
| TinyML | Tiny Machine Learning |
| DT | Decision Trees |
| ODT | Online Decision Trees |
| HT | Hoeffding Tree |
| VFDT | Very Fast Decision Tree |
| CVFDT | Concept-adapting Very Fast Decision Tree |
| HAT | Hoeffding Adaptive Tree |
| EFDT | Extremely Fast Decision Tree |
| SVFDT | Strict Very Fast Decision Tree |

# Chapter 1

# Introduction

This thesis was conducted at the Research Group on Intelligent Engineering and Computing for Advanced Innovation and Development (GECAD), focusing on Industry 4.0, a governmental initiative to digitize the economy through the Industrial Revolution. My research concentrates on modern data processing techniques within this framework, specifically data stream mining. This area emphasizes using small, resource-constrained devices and optimized algorithms to process real-time data, allowing adaptation to new concepts without necessitating retraining from scratch.

## 1.1 Context

This project is part of the PRODUTECH_R3 initiative, which aims to support industries in integrating and developing new technologies. The work has been funded by the European Union under the Next Generation EU framework through a grant from the Portuguese Republic's Recovery and Resilience Plan (PRR) Partnership Agreement within the scope of the PRODUTECH R3 project. Additionally, it has received Portuguese National Funds through the Portuguese Foundation for Science and Technology under project UIDP/00760/2020.

In the modern era, nearly every electronic device is connected to the internet, intentionally or not. The Internet of Things (IoT) extends far beyond smartphones and computers, encompassing even household appliances like vacuum cleaners, which now collect and process data. The continual evolution of IoT facilitates the optimization of diverse processes across various fields, as the constant data production from various devices, when processed, can yield valuable insights to enhance these activities.

To handle and process this vast amount of data, devices span across the Cloud and Edge layers. Ubiquitous devices, characterized by their small size, battery-powered nature, and constrained performance capabilities, operate in the edge layer. These devices are developed as specialized solutions with well-defined purposes, capable of performing their functions efficiently despite limited resources. Ubiquitous

computing integrates these capabilities into everyday objects and environments, making them interconnected and 'smart'. These devices emphasize scalability, energy efficiency, compactness, and high integrability, often employing context-aware stream mining algorithms designed for dynamic mobile environments.

Given the massive volumes of data generated, powerful data processing devices are essential for learning and acting on that data. Historically, only servers possessed the necessary computing power, but technological advancements have produced smaller, more powerful chips which allow smaller devices to have competitive advantages for certain applications. In scenarios where critical decisions must be made instantaneously, network delays in sending data to a server are impractical. The shift towards edge devices for some processing tasks has spurred further research, leading to improved components and refined algorithms for resource-constrained devices, effectively balancing the trade-off between response time and accuracy. For instance, machine learning techniques have been effectively employed for fault detection and prognosis in manufacturing equipment, utilizing data-driven models that perform well even in resource-constrained environments [20].

Although cloud computing remains a cornerstone of data processing, it is not always accessible, and transmitting collected data to a central server can compromise user privacy. Edge computing addresses these concerns by processing data locally, improving confidentiality and safety, and eliminating network delays. However, this paradigm shift necessitates a rethinking of models to maximize their potential within the constraints of memory, energy, and processing speed while maintaining accuracy.

The integration of edge computing has significantly advanced IoT environments, allowing for greater optimizations and more informed decision-making. For instance, in smart farming, smart surveillance, and lifestyle products like smartwatches, information is processed locally to derive valuable insights, such as predicting the weather, identify suspicious behavior, or analyzing sleep patterns from heart rate data. The rapid expansion of IoT has transformed data generation and processing paradigms. Unlike traditional datasets, IoT applications generate continuous, automatic data feeds requiring real-time processing and immediate decision-making capabilities. This motivated a lot of researchers to pursue this area, culminating in improving the state-of-the-art models, like neural networks and random forests, that would operate offline on high-performing servers to be now able to operate online on very constrained devices.

## 1.2 Motivation

All these evolutions in devices capabilities had to be backed by algorithm improvements as well. Although edge and ubiquitous also underwent significant advancements, they are typically developed with a well defined purposed to extract the maximum efficiency of it's components. Hence, surged the field of Tiny Machine Learning (TinyML) in 2019, which aimed at developing and applying algorithms in these devices. TinyML frameworks aim to provide low latency, optimize bandwidth utilization, enhance

safety, and reduce costs, where devices follow stringent restrictions, such as minimal power consumption, limited memory, and low cost, making it a pivotal technology for ubiquitous computing.

Understanding how to optimize algorithms like decision trees and neural networks through incremental learning, quantization, neural architecture search, and pruning is crucial for meeting the challenges of edge computing. These algorithms' ability to adapt and learn incrementally makes them suitable for data stream mining, especially in resource-constrained environments.

We aim to research how the data processing ecosystem works and how edge nodes can be enhanced. There are four core aspects to enhance, as shown in figure 1.1. We will focus on the algorithmic part, assessing what adaptations were done to decision trees to make them valuable in data stream mining and what further optimizations were done.

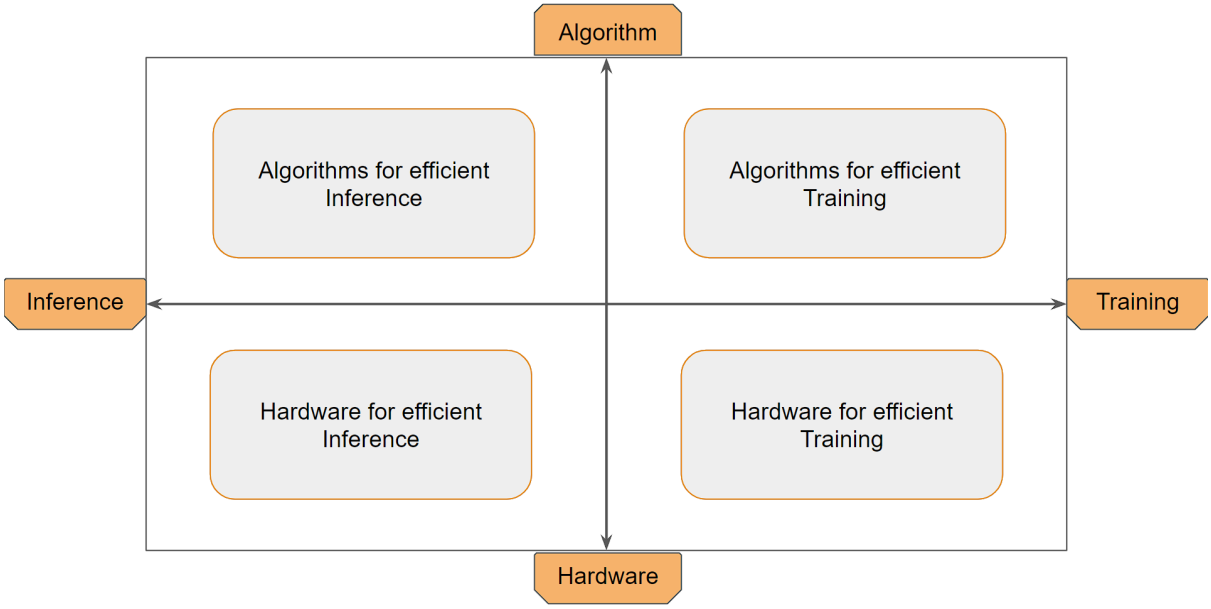


Figure 1.1: Edge Computing Optimization

## 1.3 Objectives

The primary objective of this thesis is to analyze the causes and consequences of integrating IoT applications with AI technologies. Understanding these dynamics is crucial for evaluating how IoT data generation impacts AI model development and how AI can enhance IoT functionality. Furthermore, the thesis investigates how decision tree algorithms can be adapted for real-time data stream processing, given the continuous and rapid data feeds typical of IoT environments. Identifying the strengths and limitations of decision trees in these contexts will provide valuable insights into their applicability for real-time decision-making.

In addition, this research seeks to uncover optimization techniques that enable decision trees to operate efficiently on devices with limited memory resources. Exploring methods to reduce decision tree algorithms' computational and memory footprint is essential for their deployment in ubiquitous computing devices. Finally, the thesis aims to determine the most appropriate evaluation metrics for assessing the performance of online decision trees and to identify practical applications where these optimized algorithms can be effectively utilized. This involves examining accuracy, speed, and efficiency metrics and aligning them with real-world IoT applications such as smart farming, surveillance, and wearable technologies.

The research questions guiding this study are:

1. How is data processing workflow in Internet of Things?
2. How are Decision Trees adapted for real-time data stream processing? Strengths and limitations?
3. What techniques are used to optimize Decision Trees for environments with limited memory resources?
4. What are the most suitable evaluation metrics and applications of Online Decision Trees?

## 1.4 Document Structure

We present a visual representation of the document structure in figure 1.2.

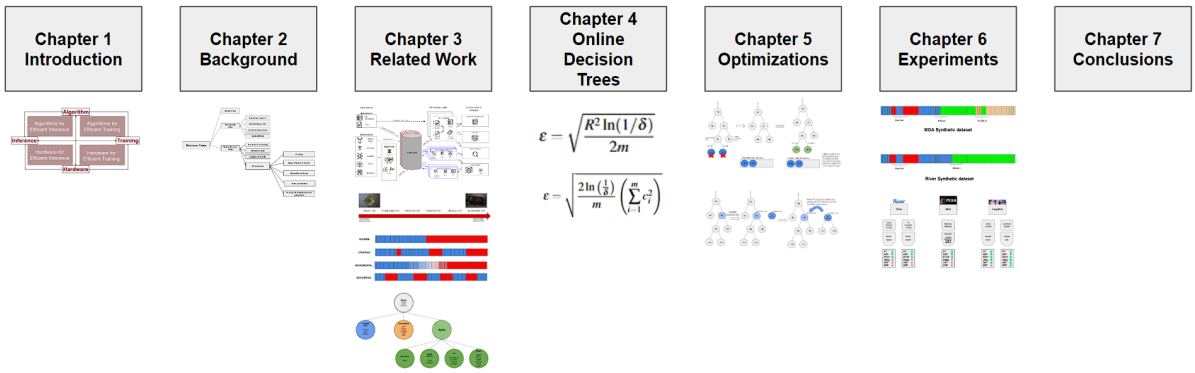


Figure 1.2: Document Structure

# Chapter 2

# Background

Decision Trees (DTs) are a cornerstone of machine learning, widely used for classification and regression tasks due to their simplicity, interpretability, and effectiveness. This section provides an overview of the significant advancements in decision trees, presented in figure 2.1, setting the stage for a detailed exploration of online decision trees.

Model Trees [16] extend traditional decision trees by incorporating more complex models at the leaves instead of simple constant values. This allows for more precise predictions by fitting linear regressions or other models within each data partition.

The group of Non-greedy Trees [16] introduces several optimization techniques. Lookahead Search improves decision trees by considering the impact of multiple future splits when selecting the current split, potentially leading to more globally optimal trees. Evolutionary Trees are constructed using evolutionary algorithms that simulate natural selection to optimize the tree structure and parameters over successive generations. Gradient-based Trees, inspired by techniques from neural networks, use gradient descent optimization to fine-tune the tree's parameters, often resulting in more accurate and deeper trees. Optimal Trees are designed to find the best possible tree structure by searching the entire space of possible trees, though they are computationally intensive.

This thesis focuses on Online Decision Trees, specialized for scenarios where data arrives continuously, and the model needs to update in real time. Incremental learning methods allow the model to update its structure and parameters as new data becomes available without retraining from scratch. Decision trees designed for streaming data can handle large volumes of continuously incoming data, making real-time predictions and updates feasible.

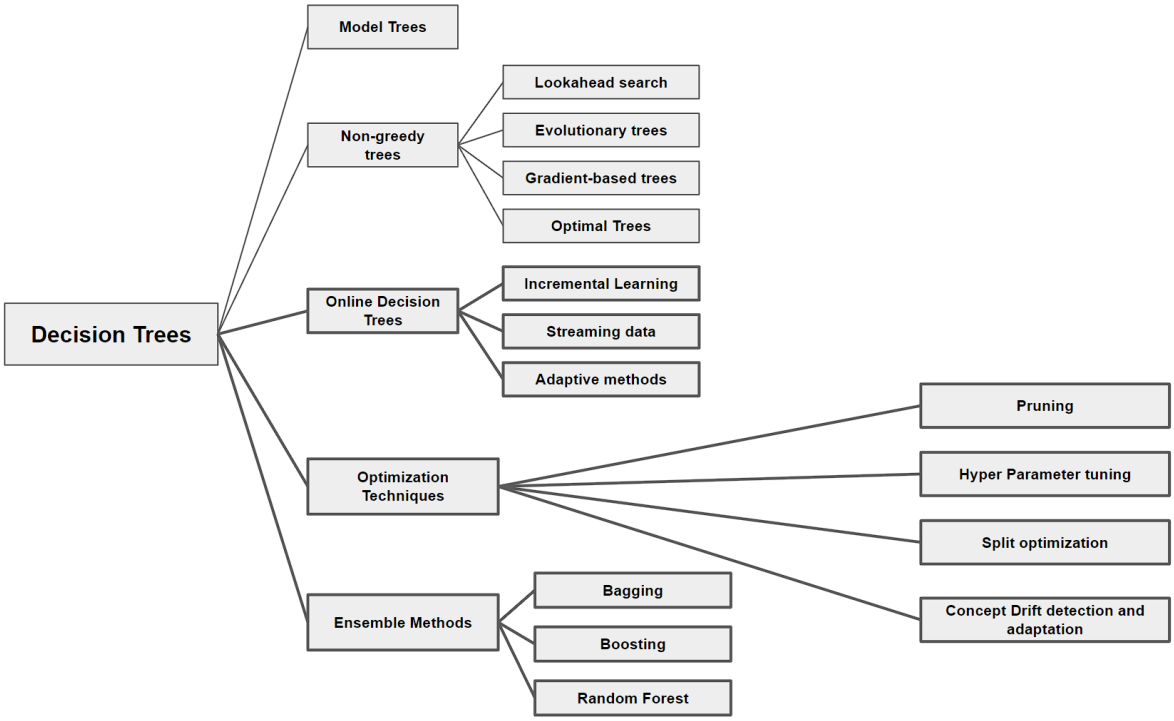


Figure 2.1: Decision Trees

## 2.1 Decision Tree

DTs are a versatile machine learning method, first introduced in the 1960s [45], allowing for classification and regression tasks. This model is highly valued for its interpretability and simplicity, as it visually represents decisions and decision-making in a tree-like model. Each node in the tree represents a feature, each branch represents a decision rule, and each leaf node represents an outcome. Decision Trees can handle categorical and numerical data and address various data-specific issues like missing values, imbalanced datasets, and small dataset sizes.

Decision trees split nodes based on feature values to create branches, aiming to maximize the homogeneity of the resulting subsets. The splitting process involves selecting the feature and corresponding threshold that best separates the data according to a chosen metric. There are various metrics including, Information Gain, Gini Impurity and Chi-Square.

The formula for entropy is given by:

$$Entropy = -\sum_{i=1}^{n} p_i \log_2(p_i) \tag{2.1}$$

where $p_i$ is the probability of occurrence of the $i$-th event and $n$ is the total number of events.

The formula for Information Gain (IG) is given by:

$$IG = Entropy(\text{parent}) - \sum_{i=1}^{k} \frac{|child_i|}{|parent|} Entropy(\text{child}_i) \tag{2.2}$$

where:

- $Entropy(\text{parent})$ is the entropy of the parent node,
- $k$ is the number of child nodes,
- $|child_i|$ is the number of instances in the $i$-th child node,
- $|parent|$ is the number of instances in the parent node,
- $Entropy(\text{child}_i)$ is the entropy of the $i$-th child node.

Regarding leaf classification, two prevalent techniques are Majority Vote and Naive Bayes. Adaptive Naive Bayes further refines decision trees by dynamically selecting between Naive Bayes and Majority Vote, guided by their historical performance. Unlike Majority Vote, which only retains the count of predicted labels, Naive Bayes maintains detailed statistics of each label prediction given the corresponding feature, which leads to increased memory use.

Another core mechanism in DT is pruning [53], which is pivotal in enhancing model performance by mitigating overfitting and simplifying the model structure. Pruning removes parts of the tree that contribute little to the prediction power or are based on noisy segments of the training data. Traditionally, in batch learning scenarios, this process is applied after the tree has been fully developed, reducing tree complexity and improving generalization to unseen data. The primary rationale behind pruning is to generate a model that, while simpler, retains or even enhances its accuracy on validation or test datasets.

Pruning strategies can be broadly categorized into pre-pruning and post-pruning described in table 2.1. Pre-pruning, also known as early stopping, restricts the tree's growth at an early stage by setting criteria such as a minimum improvement threshold for splits, a maximum allowable depth for the tree, or a minimum number of samples a node must have seen to be split. This approach aims to prevent the tree from becoming overly complex by halting its expansion when further splits are unlikely to yield significant predictive benefits. Post-pruning, on the other hand, involves allowing the tree to grow to its full depth initially and then systematically removing sections of it. Techniques within this category, such as reduced error pruning and cost-complexity pruning, focus on identifying and eliminating branches that do not improve or minimally impact the model's predictive accuracy.

The dynamics of pruning in online decision trees, designed to operate in data stream environments, incorporate additional considerations due to the continuous influx of data and the potential for evolving data distributions, known as concept drift. In such settings, pruning is not merely a mechanism for simplification but also an adaptive process that ensures the tree remains relevant as new data arrives. This necessitates dynamic and adaptive pruning techniques integrating change detection mechanisms like ADaptive WINdowing ADWIN [5]. These techniques enable the tree to identify significant shifts

Table 2.1: Comparison of Pre-Pruning and Post-Pruning Techniques

| **Technique** | **Pre-Pruning** | **Post-Pruning** |
|---|---|---|
| **Definition** | Pruning occurs during the tree building process. | Pruning occurs after the tree is fully grown. |
| **Examples** | Set a maximum depth for the tree, stop splitting if the gain is below a threshold, require a minimum number of samples for a split. | Reduced error pruning, cost-complexity pruning (e.g., CART pruning). |
| **Advantages** | Prevents overfitting early, faster training times due to smaller trees. | Often results in more accurate models, can simplify overly complex trees after training. |
| **Disadvantages** | Risk of underfitting if stopped too early. | Requires full tree construction initially, which can be computationally expensive. |

in the data distribution, allowing for the pruning of branches that become irrelevant or misaligned with the current trends.

## 2.2 Decision Tree vs Neural Networks

In the realm of tabular data, DTs and their ensemble methods have demonstrated superior performance to Neural Networks (NNs) [10, 28, 13], attributed to several key factors described in table 2.2.

Firstly, real-world data often contains missing values, outliers, and class imbalances. DT can handle these issues by approximating missing values and managing variable ranges during the split process. At the same time, NNs require extensive preprocessing to address the irregularities, which makes their application to tabular data more demanding.

Secondly, tabular data generally lack spatial correlations among variables, and the relationships between features tend to be complex and irregular. When NNs are trained from scratch on tabular data, they often form overly simplistic decision boundaries due to simplicity bias, rendering them less effective at capturing intricate relationships. In contrast, DTs are not subject to this bias and are better suited to adapt to the complex dependencies in tabular datasets.

Another key factor is the reliance on preprocessing. Managing categorical features poses a particular challenge for NNs. Techniques such as one-hot encoding can create sparse feature matrices, while ordinal encoding can introduce artificial orderings. These preprocessing steps can diminish the advantages of NNs' inherent representation learning. Conversely, DTs can effectively manage varying feature importance by selecting relevant features and appropriate thresholds without extensive preprocessing.

Moreover, even the slightest change in categorical or binary features can significantly impact predictions in tabular data. DTs handle this variability effectively by concentrating on the most relevant features and disregarding less important ones, resulting in more accurate and interpretable models. NNs, on the other hand, often struggle with this issue unless they undergo substantial tuning and architectural adjustments. DTs are preferred not only for their robustness but also for their simplicity and interpretability. They demand fewer computational resources and preprocessing steps, making them easier to implement and understand. Their clear if-else decision paths minimize computational overhead and simplify the parameterization process, which is particularly beneficial when working with real-world data that may be incomplete or messy.

## 2.3 Application Contexts

In the rapidly evolving landscape of IoT, innovative data engineering strategies require innovative data engineering strategies to address the unique challenges posed by IoT applications, such as sensor data management, operational technology (OT) and information technology (IT) integration, and the proliferation of connected smart devices [34, 49]. These strategies must efficiently capture, process, and store data to make it available for analysis and machine learning (ML). The resultant shift in data processing approaches leads to various consequences, impacting the design and implementation of IoT systems [47].

The emergence of edge computing has also redefined the resource paradigms for AI in IoT applications. AI resources now span a spectrum from cloud-based machine learning (Cloud ML) with extensive computational power to mobile machine learning (Mobile ML) and TinyML, designed for devices with severe memory and power constraints. Each paradigm addresses specific IoT ecosystem needs, optimizing for accuracy, efficiency, and energy consumption. TinyML, in particular, is critical for ultra-low-power AI applications, enabling intelligent processing on devices with limited resources, often powered by small batteries.

The following sections explore various application contexts where these innovative algorithms and strategies can be applied, demonstrating the transformative power of IoT technologies across different sectors.

### 2.3.1 Industrial IoT

Industrial IoT (IIoT) involves using IoT technologies to enhance operational efficiency, safety, and productivity in manufacturing and industrial settings. IIoT systems connect machines, sensors, and devices to collect and analyze data, providing valuable insights into industrial processes.

**Predictive Maintenance:** IIoT sensors monitor the condition of machinery, including vibration, temperature, and sound levels. Online decision trees analyze this data to predict potential equipment failures

before they occur. For example, abnormal vibration patterns can indicate a bearing failure, prompting maintenance actions before a breakdown happens [47].

**Operational Optimization:** By continuously analyzing data from various machines and processes, IIoT systems can optimize production workflows. For instance, adjusting machine settings in real-time to maintain optimal performance and reduce downtime.

### 2.3.2 Mining Query Streams

Searching the web for information has become a vital part of everyday life. Search engines like Google, Bing, and Yahoo process millions of queries daily. Consequently, significant research has focused on mining these query streams to enhance the quality and relevance of search results for users.

**Query Clustering:** Facilitates quick browsing by clustering search results. For example, Zeng et al. [61] focused on clustering web search results to aid user navigation.

**Temporal Correlation Analysis:** Enhances search accuracy by analyzing temporal patterns in queries. Chien and Immorlica [14] improved search results by suggesting alternative queries based on temporal correlations.

### 2.3.3 Network Monitoring

The internet comprises numerous interconnected routers that communicate via IP packets. Effective network management involves real-time traffic data analysis.

**Traffic Data Logging:** Traffic data is recorded at various levels, including packet logs (source/destination IP addresses), flow logs (packet counts, start/end times, protocols), and SNMP logs (aggregated byte counts) [44].

**Real-time Classification:** Identifying and preventing malicious attacks (e.g., DOS, R2L, U2R, probing attacks) using real-time data stream classifiers is essential for maintaining network security and performance.

### 2.3.4 Sensor Networks

Sensor networks consist of spatially distributed sensors that monitor environmental conditions (e.g., temperature, sound, vibration). These networks are critical in applications like traffic monitoring, smart homes, habitat monitoring, and healthcare.

**Real-time Data Processing:** Sensor networks generate massive data streams that need to be processed in real time to extract meaningful information.

**Healthcare Applications:** In modern hospitals, sensor networks monitor physiological data of patients. Systems must analyze this data to identify significant clinical events and support medical decision-making, as emphasized in studies like [47].

### 2.3.5 Social Network Streams

Online social networks (e.g., Facebook, Twitter, LinkedIn) produce vast amounts of data, including text, multimedia, and interaction logs.

**Stream Clustering:** Detecting and monitoring community evolution within social networks to understand how communities emerge, grow, and evolve [1, 36].

**Stream Classification:** Classifying users, categorizing discussion topics, and detecting events through real-time analysis of social network data streams [51, 2].

Each of these application contexts demonstrates the transformative power of IoT technologies across different sectors, highlighting their potential to improve efficiency, safety, and outcomes through real-time data analysis and automation.

Table 2.2: Comparison of Neural Networks and Online Decision Trees

| Aspect | Neural Networks (NNs) | Online Decision Trees (ODTs) |
|---|---|---|
| **Handling Missing Values** | Requires extensive preprocessing and imputation techniques. | Can approximate missing values during the split process. |
| **Dealing with Outliers** | Sensitive to outliers; requires normalization or standardization. | More robust to outliers due to split-based decision making. |
| **Class Imbalance** | Requires techniques such as resampling, weighting, or synthetic data generation. | Can handle class imbalances inherently through the splitting criteria. |
| **Preprocessing** | Requires extensive preprocessing, especially for categorical features (e.g., one-hot encoding). | Minimal preprocessing needed; handles categorical and continuous features natively. |
| **Training Complexity** | High computational demand, requires extensive tuning and often large datasets. | Lower computational demand, simpler to train with fewer hyperparameters. |
| **Interpretability** | Generally considered a "black box" model with limited interpretability. | High interpretability with clear if-else decision paths. |
| **Handling Concept Drift** | Requires sophisticated mechanisms for detecting and adapting to concept drift. | Can incorporate dynamic pruning and change detection mechanisms to handle concept drift effectively. |
| **Computational Resources** | High computational resources required for training and inference. | Relatively low computational resources needed, suitable for real-time applications. |
| **Complex Decision Boundaries** | Excels at modeling complex decision boundaries, combining multiple features to define a boundary | May struggle with complex decision boundaries using just one feature to split, requiring more splits or deeper trees. |
| **Scalability** | Highly scalable, especially with modern deep learning frameworks and hardware acceleration (GPUs, TPUs). | Scalability can be challenging with very large datasets or deep trees. |
| **Adaptability** | Can adapt to a wide range of tasks (e.g., classification, regression, image recognition, natural language processing). | Typically more specialized for structured data and tabular datasets. |
| **Transfer Learning** | Capable of leveraging pre-trained models to transfer knowledge to new, related tasks. | No straightforward mechanism for transfer learning; each tree is trained from scratch. |

# Chapter 3

# Related Work

In this chapter, we delve into the landscape of data processing and decision tree algorithms, focusing on their applications and advancements in modern IoT environments. We begin by exploring the workflow of data processing, highlighting the pivotal role of the Event Hub and the transition to edge computing. We then examine the integration of streaming and microservice platforms, addressing the balance between batch and stream processing. Furthermore, we review the core assumptions and challenges of data stream mining, including concept drift and incremental learning. The chapter also presents key surveys on decision trees, detailing recent advancements, pruning techniques, and various Hoeffding Tree algorithms. Finally, we provide a comprehensive overview of significant advancements in online decision trees, illustrating their evolution and impact on real-time data analysis and decision-making.

## 3.1 Data Processing

In modern IoT environments, the workflow of data processing begins at the source, where data is continuously generated by various devices, such as mobile apps, IoT sensors, location tracking systems, social media platforms, and telemetry units. These diverse sources produce vast amounts of data that need to be collected, processed, and analyzed in a timely and efficient manner. In figure 3.1, we have a visual representation of the data processing workflow [34, 49].

### 3.1.1 Event Hub: The Central Buffer

At the heart of this data processing workflow is the Event Hub, depicted in red in the figure 3.1. This crucial component acts as a central buffer, ingesting data from all connected sources. The Event Hub's primary role is to manage the continuous flow of incoming data streams, ensuring that downstream systems are not overwhelmed by sudden spikes in data volume [3, 47]. By temporarily storing the data, the Event Hub allows for a smoother, more controlled data processing pipeline.

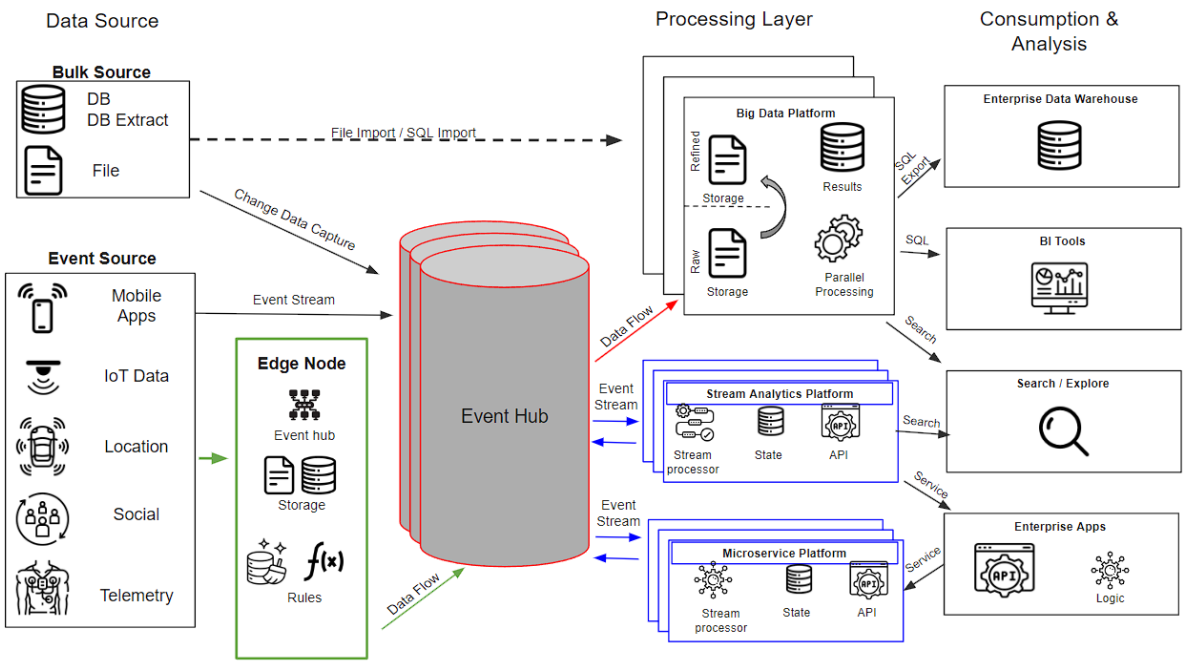


Figure 3.1: Data Processing Schema

As data flows into the Event Hub, it is organized into event streams, which are then directed towards different processing platforms. This initial step is vital for maintaining the integrity and manageability of the data as it moves through the system [34].

### 3.1.2 Transition to Edge Computing

Traditionally, data collected by the Event Hub would be sent to centralized servers or cloud platforms for processing, yet this approach often encounters significant delays due to network latency and bandwidth limitations [38]. As the volume of data generated by IoT devices continues to grow, the need for faster and more efficient data processing becomes critical [12].

Edge computing has emerged as a viable solution to address these challenges, processing data closer to its source rather than relying on centralized data centers [54]. This paradigm shift is facilitated by ubiquitous computing, where devices that only collect and transmit data start performing some level of data processing. These devices, also known as edge nodes, reduce the load on central systems by handling data processing tasks locally, enabling faster decision-making and improved performance [57].

Edge nodes are depicted in green in figure 3.1 and include various categories of edge devices, as shown in Figure 3.2. The lower-tier devices can benefit from the TinyML field. It focuses on developing machine learning algorithms for devices with small form factors and low power consumption. TinyML devices typically operate at clock speeds below 100MHz with about 256KB of RAM, while those clocked above 100MHz feature RAM capacities ranging from 256KB to 8MB [52]. These specifications highlight the

need for highly efficient algorithms that operate within tight memory limits and low power budgets. Most of them are also designed to be powered by coin or Li-Po batteries and require power consumption in the milliwatt (mW) range, making them ideal for battery-operated IoT devices.

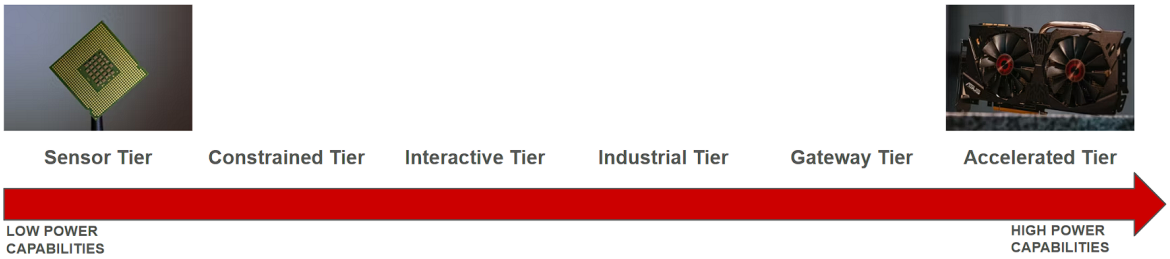


Figure 3.2: Edge Devices

The evolution from cloud-based machine learning (Cloud ML) to TinyML represents a significant shift towards more decentralized and resource-efficient machine learning applications. This transition is crucial for advancing the capabilities of IoT systems, ensuring that even with limited computational resources, these systems can perform complex tasks effectively and efficiently [19].

The development and optimization of algorithms tailored to the constraints of TinyML devices are essential for enhancing the performance of IoT systems [10]. This evolution underscores the importance of designing models that are not only accurate but also resource-efficient, promoting advances IoT and edge computing technologies. benczur2018online

### 3.1.3 Streaming and Microservice Platforms

Once data is buffered in the Event Hub, it is routed to various processing platforms, highlighted in blue in figure 3.1. These platforms, which include the Stream Analytics Platform and the Microservice Platform, are essential for achieving efficient, real-time data processing while maintaining a modular and scalable architecture [63].

The Stream Analytics Platform is designed for real-time data processing, continuously analyzing event streams to provide immediate insights and trigger necessary actions. This capability is particularly effective for detecting anomalies, monitoring system health, and responding to critical events [25, 47]. By leveraging frameworks such as Apache Kafka Streams, the platform enables timely responses to data trends, which is crucial for managing the high volumes and velocities typical of IoT environments. Moreover, its ability to scale horizontally by adding more stream processors ensures that it can handle increased data loads efficiently. The flexibility gained by implementing stream processing applications as microservices further enhances the system, ensuring that updates or changes to one service do not disrupt the entire system [29].

Simultaneously, the Microservice Platform decomposes data processing tasks into smaller, independent services, each handling a specific function. This modular approach allows for a highly scalable architecture that facilitates maintenance and updates without disrupting the entire system. The loose coupling of microservices, where each focuses on a single task, makes them easily changeable or replaceable, thus enhancing the system's adaptability to evolving requirements [34]. Furthermore, the independent development, deployment, and scaling of microservices enable faster updates and efficient resource utilization. The use of container technologies like Docker adds to their fast and flexible deployment, ensuring consistency across different environments.

The integration of these platforms involves a seamless flow of data. IoT devices continuously generate data, which is ingested by the Event Hub, organized into event streams, and directed towards the Stream Analytics Platform. Here, stream processors perform real-time analytics and generate immediate insights. These data processing tasks are then divided among various microservices within the Microservice Platform, which handle functions such as data transformation, aggregation, and anomaly detection. Processed data is pushed back into the log data store as new topics, enabling near real-time querying and analysis through the analytics data store. This ensures that the processed data is readily available for further use by enterprise applications, BI tools, and other endpoints [34].

### 3.1.4 Balancing Batch and Stream Processing

Despite the advantages of real-time processing, there are scenarios where batch processing remains relevant. Batch processing involves collecting data over a period and processing it in large chunks at scheduled intervals, which is effective for comprehensive data analysis, focusing on understanding long-term patterns and trends rather than immediate responses.

However, batch processing introduces high latency, by the time the data is analyzed it may already be outdated. To address this, many systems adopt a hybrid approach, combining the strengths of both batch and stream processing, exemplified by the Lambda Architecture [56], which integrates batch processing for historical data analysis with stream processing for real-time insights.

In this hybrid model, raw data is ingested in real-time and fed into both batch and stream processing layers. The batch layer ensures accuracy and completeness by processing data in large chunks, while the stream layer provides immediate updates and insights. This dual approach allows for robust and timely data processing, catering to both real-time and historical analysis needs.

### 3.1.5 Stream Processing and Data Streams

In a streaming architecture, analytics are performed on each event as it arrives, allowing for real-time decision-making and immediate action based on the data [63]. For example, if an anomaly is detected, an event can be generated and sent back to the event stream for further processing or action by another

service. They typically consist of multiple services rather than a single monolithic system. This microservice approach organizes the architecture into independent modules, each handling specific tasks. This separation of concerns is crucial for scalability and maintainability. Microservices can consume data from event tables and produce new events back to the event stream, allowing for flexible and scalable data processing.

Data streams are characterized by a continuous flow of information with varying volume, velocity, and variability [3]. Unlike conventional data mining, where models are trained on a complete dataset, data stream mining deals with the ongoing influx of data. This approach is critical for environments where data is abundant and rapidly changing, such as financial tickers, sensor networks, and social media feeds. Traditional batch methods become obsolete under these conditions due to the sheer volume and speed of incoming data.

Data stream mining operates under four core assumptions:

1. Process each example at a time and inspect it only once.
2. Use a limited amount of memory.
3. Work within a limited amount of time while allowing the model to incorporate new information as it arrives.
4. Be ready to predict at any point.

Incremental learning is essential for models in the data stream mining scenario. It's the capacity of the model to update as new data arrives without retraining from scratch. Within TinyML, applying this concept to restrained devices implies more thoughtful ways to manage memory and computing demands. We can apply pruning techniques to allow decision trees to manage their size. By pruning, decision trees can adapt to memory constraints, reducing its size if necessary and removing subtrees that represent noise or an old concept [53].

Another big factor in data streams is Concept Drift. Concept drift refers to the changes in the statistical properties of the target variable over time, which can degrade model accuracy and be managed by detection and adaptation strategies [40]. We consider various drift methods, which are presented in figure 3.3, from which we will apply two to generate custom datasets with concept drift and test model adaptability.

### 3.1.6 Bottlenecks and Challenges

Managing the continuous flow of data in IoT applications presents several challenges. Traditional database systems often struggle with high data rates and fluctuating volumes, a characteristic of IoT environments [34, 49]. The Event Hub provides a buffering solution but introduces latency as data is processed in batches.

Streaming architectures, while addressing the latency issue by processing data as it arrives, require robust systems capable of handling continuous data flows. The pub-sub model, where data producers

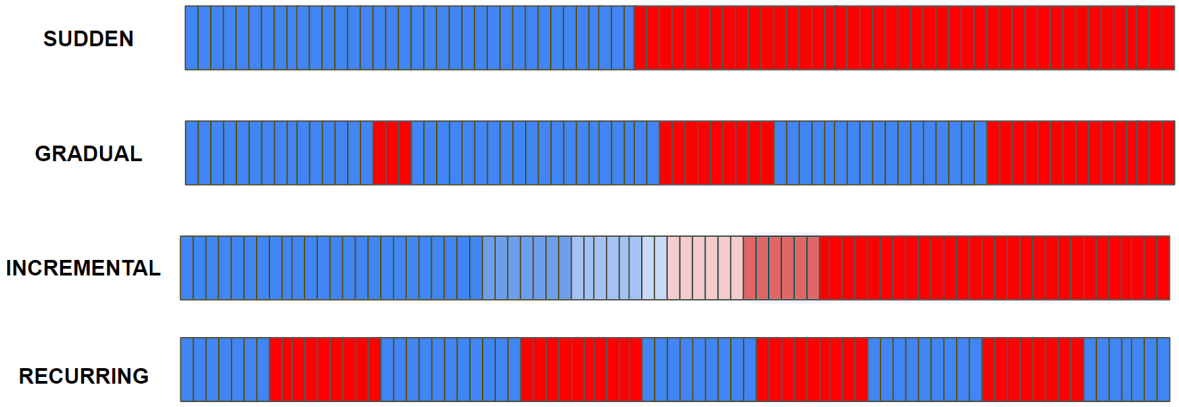


Figure 3.3: Data Processing Schema

publish events to topics and consumers subscribe to these topics, enables asynchronous communication and decouples data sources from data consumers. This model is effective for managing high volumes of data but requires efficient indexing, storage, and querying techniques to ensure timely data processing and retrieval.

## 3.2 Surveys

We identified four key surveys focused on decision trees. One outlines recent advancements in decision trees, encompassing both batch and streaming applications, another explores various pruning techniques used in decision tree methodologies, and the last two specifically address Online Decision Trees, with a particular emphasis on Hoeffding Trees. The articles were selected based on relevance to the field, citation frequency, and impact factor of the publishing journals. Databases such as IEEE Xplore, ACM Digital Library, ResearchGate, and ScienceDirect were extensively searched using keywords related to decision trees, pruning techniques, and data streams. The inclusion criteria were confined to peer-reviewed journal articles and conference papers published in English. This comprehensive approach ensures a robust and insightful synthesis of state-of-the-art decision tree algorithms.

This comprehensive survey [16] thoroughly examines the latest advancements and extensions in decision tree algorithms over the past decade. The paper primarily contributes to my research by detailing various optimization techniques for decision trees, which are crucial for environments with limited memory resources. Key techniques discussed include cost-sensitive learning, soft decision trees, gradient-based trees, evolutionary trees, and optimal decision trees. Additionally, the survey outlines suitable evaluation metrics, such as accuracy, memory usage, computation time, and energy efficiency, essential for assessing decision tree performance in these environments. While the paper does not explicitly address real-time data stream processing, the insights provided into optimization techniques and practical applications significantly support the objectives of my research, particularly in understanding how to optimize

decision trees for limited memory resources.

Shamrat's paper [53] extensively analyzes various pruning techniques used to optimize decision trees, presenting pre and post-pruning methods. The study evaluates methods like Chi-square pruning, SVM pre-pruning, reduced error pruning, minimum error pruning, and cost-complexity pruning, demonstrating their effectiveness in reducing overfitting and improving generalization. The relevance of this paper to my research lies in its detailed exploration of pruning as a technique for optimizing decision trees in memory-constrained environments. By reducing the size and complexity of decision trees, pruning methods make them more efficient and suitable for deployment in limited-memory settings. Additionally, the paper discusses suitable evaluation metrics such as accuracy, tree size, and classification performance, providing empirical results that highlight the practical applications of pruned decision trees.

The paper [46] discusses several variants of the Hoeffding Tree algorithm, each tailored to optimize performance in different scenarios. These include single trees, decision trees based on the Hoeffding bound, window-based Hoeffding Trees, weight-based Hoeffding Trees, distribution-based Hoeffding Trees, and ensembles of Hoeffding Trees. These models address various aspects of data distribution and concept drift, enhancing their applicability in memory-constrained environments. The paper highlights that Hoeffding Trees provide robust performance guarantees, a feature not shared by other incremental decision tree learners. This robustness, combined with their ability to adapt to concept drift, makes them suitable for anomaly detection in streaming data, including applications such as intrusion detection and fault detection in dynamic environments. This survey emphasizes exploring ensemble methods while we pretend to present the various single Hoeffding Tree implementations, which can then be used with bagging or boosting techniques.

Jumar's paper [35] provides an in-depth examination of Hoeffding Trees and their application in classifying streaming data, comparing several algorithms, including the standard Hoeffding Tree (VFDT), Streaming Random Forest (SRP), and Concept Adapting Very Fast Decision Tree (CVFDT). The authors emphasize the efficiency of Hoeffding Trees in processing large data streams with limited memory, making them suitable for environments with constrained resources. They also highlight how these algorithms handle real-time data stream processing and concept drift through techniques like the Hoeffding bound and sliding window mechanisms. The survey extensively covers optimization techniques for decision trees in memory-constrained environments, discussing ensemble methods like Streaming Random Forest and dynamic tree updates in CVFDT. Evaluation metrics such as accuracy, Kappa statistic, processing time, and memory usage are used to assess the performance of these algorithms, demonstrating their practical applications in areas like network traffic classification and anomaly detection. While the paper is comprehensive, it does not specifically address the implementation of these algorithms on TinyML devices, suggesting a potential area for further research.

Table 3.1: Related Survey papers

| | RQ1 | RQ2 | RQ3 | RQ4 |
|---|---|---|---|---|
| Recent advances in decision trees: An updated survey [16]. | | | X | |
| A comprehensive study on pre-pruning and post-pruning methods of decision tree classification algorithm[53]. | | | X | |
| Hoeffding tree algorithms for anomaly detection in streaming datasets: A survey [46]. | | X | X | X |
| A survey on Hoeffding tree stream data classification algorithms[35]. | | X | X | |

## 3.3 Online Decision Trees

This section reviews significant advancements in the development of Online Decision Trees ordered by complexity, starting by presenting the most basic and foundational algorithms and proceeding to present more complex extensions that explore the limitations of previously implemented trees. We categorize the trees presented or referenced in this thesis in figure 3.4 and present them chronologically in table 3.2.

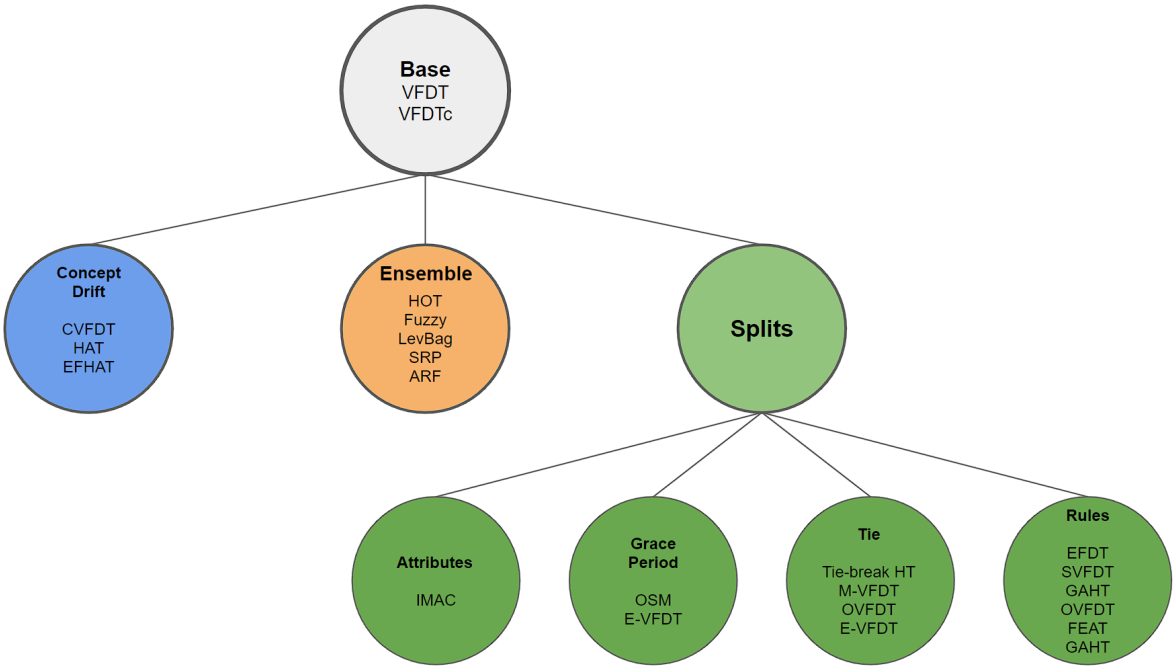


Figure 3.4: Hoeffding Tree by category of optimization

More than four decades ago, Breiman et al. published "Classification and Regression Trees" (CART)

Table 3.2: Algorithm Timeline

| **Algorithm Name** | **Year** | **Reference** | **Description** |
|---|---|---|---|
| VFDT | 2000 | [17] | Base Implementation |
| CVFDT | 2001 | [33] | Concept Drift Adaptation (Fixed-Window) |
| VFDTc | 2003 | [22] | Naive Bayes classifier and Gaussian Splitter implementation |
| Tie Breaking in HT | 2005 | [32] | Tie breaking improvements |
| HOT | 2007 | [48] | New option Tree (ensemble like) |
| HAT | 2009 | [4] | Concept Drift Adaptation (Adaptive Window) |
| LevBag | 2010 | [6] | Ensemble |
| M-VFDT | 2011 | [58] | Adaptive Tie-Treshold |
| OVFDT | 2011 | [60] | Adaptive Tie and pruning rules |
| McDiarmid Bound | 2012 | [50] | Alternative to Hoeffding Bound |
| ARF | 2017 | [26] | Ensemble |
| EFDT | 2018 | [42] | Relaxed splitting |
| SFVDT | 2018 | [15] | Stricter splittling |
| OSM | 2018 | [39] | Dynamic split reevaluation period |
| SRP | 2019 | [27] | Ensemble |
| FEAT | 2020 | [62] | Fair-Enhancing Information Gain and concept drift adaptation |
| GAHT | 2021 | [24] | Alternative expansion method |
| Fuzzy | 2021 | [18] | Fuzzy Information Gain and Fuzzy sets (ensemble like) |
| IMAC | 2021 | [55] | Split optimization (attributes) |
| EFHAT | 2022 | [41] | Combining EFDT with HAT |
| E-VFDT | 2022 | [37] | Adaptive Tie Treshold and Grace Period, and remove poor attributes |

[11], which introduced a powerful and intuitive method for decision tree analysis. This seminal work laid the foundation for numerous advancements in decision tree algorithms. Initially, machine learning, including decision tree methods, was predominantly applied to static datasets stored in databases. However, with the advent of big data and advancements in technology, there has been a paradigm shift towards real-time data analysis. Researchers have developed sophisticated techniques to adapt decision tree algorithms for streaming data, enabling real-time knowledge discovery and decision-making. This evolution reflects the growing need to analyze dynamic data and extract actionable insights promptly.

In 2000, Domingos and Hulten introduced the **Very Fast Decision Tree** (VFDT) [17], a groundbreaking decision tree algorithm designed to handle streaming data efficiently. VFDT leveraged the computational simplicity of decision trees with the Hoeffding bound to validate splits, allowing it to perform exceptionally well in real-time scenarios. However, VFDT struggled with streams exhibiting concept drift. To address this limitation, the same researchers developed the Concept-adapting Very Fast Decision Tree (CVFDT) [33], which incorporated a windowing method to adapt to different types of drift, thereby maintaining an up-to-date model.

While VFDT was a significant advancement, it had limitations, particularly in handling numerical attributes and fully exploiting its potential. In 2003, further enhancements introduced a continuous at-

tribute splitter and the capability to apply a Naive Bayes classifier to the tree leaves. This enhanced algorithm, known as VFDTc [22], improved the accuracy and versatility of VFDT in streaming environments.

To improve tie-breaking mechanisms in HT, Holmes [32] introduced two extensions. The first extension validated ties by implementing a comparison rule between different splits, ensuring that ties were genuine and not artifacts of the algorithm. The second extension introduced a tie-breaking period that increased in the child nodes if the parent node encountered a tie, thus preventing repeated tie-breaking scenarios and improving the overall decision process.

In 2007, Kirkby proposed the Hoeffding Option Tree (HOT) [48], an innovative extension of VFDTc that shared similarities with ensemble methods. HOT allowed for optional branches at each node in addition to the typical splits, leading to different leaves and providing multiple pathways to reach a decision. The final prediction was a weighted average of the results from these multiple leaves, enhancing the robustness and accuracy of the model.

The evolution continued with the Hoeffding Adaptive Tree (HAT) [4], which improved the efficiency of data storage and handling. Instead of using a fixed-size sliding window for all nodes, HAT employed the ADWIN (Adaptive Windowing) change detector to adjust window sizes for each node dynamically. This approach allowed HAT to store all statistical data in main memory, avoiding the need for disk storage like did CVFDT for large window sizes. Additionally, ADWIN detected changes and triggered the training of alternate trees on faulty nodes, enhancing the tree's adaptability to concept drift.

In 2011, Moderate VFDT (M-VFDT) [59] was introduced to enhance split tie-breaking. This algorithm proposed an adaptive tie-breaking threshold, calculated periodically using the dynamic mean of the Hoeffding Bound (HB). This adaptive approach ensured the tie-breaking mechanism remained in tune with the incoming data, improving decision accuracy.

Extremely Fast Decision Tree (EFDT) [42] garnered significant attention due to its rapid learning capabilities. Unlike VFDT, which expended considerable computing power attempting multiple splits at a node, EFDT opted to split more frequently when an attribute's gain was above zero. EFDT also revisited nodes to verify the optimality of previous splits, pruning suboptimal subtrees and creating new splits based on current best attributes. Although not explicitly designed for drift detection, EFDT's reevaluation technique provided a degree of drift tolerance.

Strictly Very Fast Decision Tree (SVFDT) [15] introduced novel constraints to improve tree expansion accuracy and speed. SVFDT implemented two sets of conditions: one group aimed for higher accuracy with stricter constraints, while the other allowed faster expansion with more relaxed constraints. Both conditions are built on the base VFDT split criteria, requiring the tree to collect more examples and achieve greater certainty before performing splits, thereby reducing resource expenditure on splits and ties.

One-Sided Minimum (OSM) [39] focused on dynamically adjusting the re-split period after a tie. By

analyzing class distributions from the best split at a node, OSM calculated the number of additional examples needed to meet the Hoeffding Bound. This method ensured more accurate and efficient decision-making within the algorithm, enhancing overall performance.

Researchers also explored the energy consumption of VFDT. Eva García-Martín's work in this area led to developing the Green Accelerated Hoeffding Tree (GAHT) [24]. GAHT introduced a utility metric to evaluate the usefulness of a leaf based on the examples it processed influencing the tree's growth rate, with low scores leading to temporary deactivation of leaves, moderate scores prompting growth according to VFDT criteria, and high scores accelerating expansion using EFDT tactics. This adaptive approach optimized energy consumption and computational efficiency.

Lastly, the Extremely Fast Hoeffding Adaptive Tree (EFHAT) [41] combined the change detection and structure revision capabilities of HAT with the split mechanism of EFDT. This hybrid approach leveraged ADWIN for dynamic window adjustment and incorporated EFDT's frequent splitting and reevaluation techniques, resulting in a highly adaptable and efficient decision tree algorithm.

# Chapter 4

# Online Decision Trees

This chapter introduces how decision trees were adapted to process data streams, also known as Online Decision Trees. The various aspects covered are presented in figure 4.1. Applying Incremental Learning to an algorithm allows it to process data streams by gradually learning on data, building upon past knowledge, and adapting itself to new concepts without needing to retrain from scratch like in batch methods. The tree needs to have a dynamic structure re-adjusting to changes. This adaptability can be inherent by tuning hyperparameters such as Grace Period and Tie Threshold, making the tree perform better splits and not degrade performance as much. Leaf classification (e.g., majority vote, naive bayes) and attribute handling (feature selection, outliers) can also enhance tree capabilities to endure concept drift. The main technique for dealing with concept drift is pruning, either done by split revaluation or concept drift detection at the nodes.

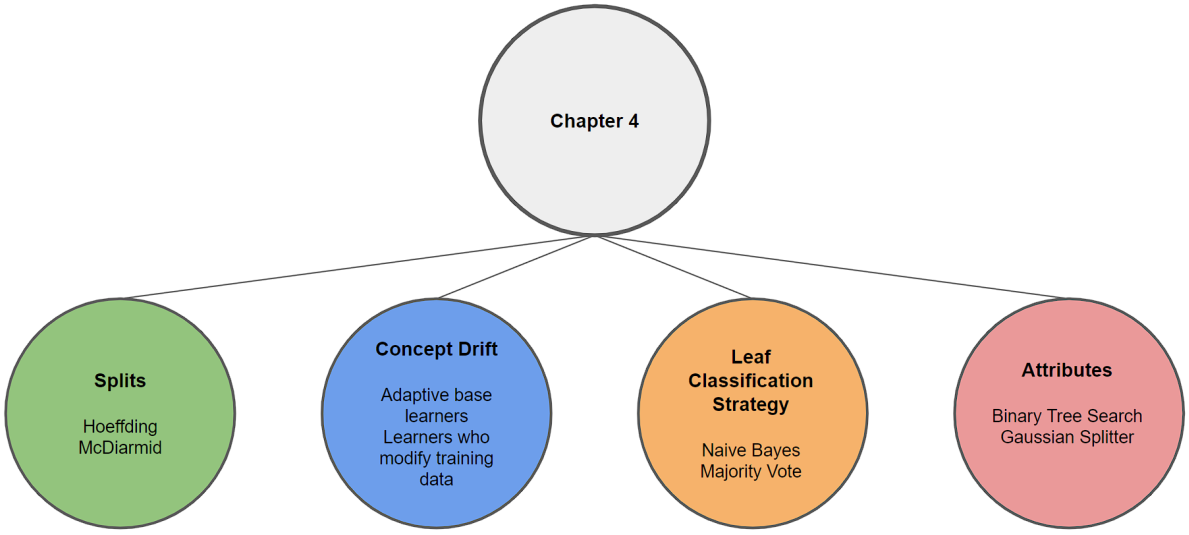


Figure 4.1: Chapter 4 Structure

## 4.1 Splits

ODTs measure split utility using a criterion, typically Information Gain, calculated by comparing the entropy before and after the split on an attribute, measuring how much information a feature gives us about the class. The criteria calculation involves updating statistical summaries of the data passing through, leveraging moving averages or online algorithms. Fuzzy Information Gain (FIG) [18] and Fair-Enhancing Information Gain (FEIG) [62] modify the calculation of traditional information gain to address specific challenges in data handling and decision fairness. FIG calculates information gain by measuring the fuzzy entropy before and after a split, where fuzzy entropy quantifies uncertainty regarding the degrees of membership of data points to fuzzy sets. This approach allows for soft thresholds in decision trees, accommodating the imprecision and vagueness inherent in many real-world datasets. On the other hand, FEIG adjusts the traditional information gain formula by multiplying it with a Fair-Enhancing Gain (FEG), which assesses the reduction in discrimination due to a potential split. Thus, FEIG emphasizes splits that reduce uncertainty and discrimination, ensuring the tree's decisions are informative and fair.

When comparing splits to ensure one is truly superior, additional statistical methods, like the Hoeffding Bound [30], support the current decision as valuable.

$$\varepsilon = \sqrt{\frac{R^2 \ln(1/\delta)}{2m}} \tag{4.1}$$

- $\varepsilon$ is the Hoeffding Bound.
- $R$ is the range of the random variable (i.e., the difference between the maximum and minimum possible values).
- $\delta$ is the probability that the true mean (or true error) is outside the $\varepsilon$-interval around the empirically observed mean (or observed error).
- $m$ is the sample size.

The Hoeffding Bound remains applicable across different datasets irrespective of their underlying distributions. The cause stems from its reliance on just the variable values range, the sample size, and the confidence level specified. The broad applicability has drawbacks since the bound is often more conservative than bounds tailored for specific distributions. Such conservatism robustness may limit sensitivity in cases where detailed distributional information is available.

An alternative to Hoeffding Bound is McDiarmid's inequality [50], a concentration measure used to bound the deviation of a function of independent random variables from its expected value. It provides a way to determine how much the function value can deviate from the expected value with high probability.

$$\varepsilon = \sqrt{\frac{2\ln\left(\frac{1}{\delta}\right)}{m}\left(\sum_{i=1}^{m} c_i^2\right)} \tag{4.2}$$

- $\varepsilon$ is the McDiarmid bound.
- $c_i$ represents the bounded differences, such that $|f(Z) - f(Z')| \leq c_i$ for the $i$-th component.
- $\delta$ is the probability that the true mean (or true error) is outside the $\varepsilon$-interval around the empirically observed mean (or observed error).
- $m$ is the sample size.

$$\sum_{i=1}^{m} c_i^2 \tag{4.3}$$

- $c_i$: This is the bounded difference for the $i$-th component. It quantifies how much the function $f(Z)$ can change when the $i$-th variable in the dataset is altered while keeping all other variables fixed.
- $c_i^2$: This is the square of the bounded difference for the $i$-th component.
- $\sum_{i=1}^{m} c_i^2$: This is the sum of the squared bounded differences over all $m$ components.

In the context of McDiarmid's inequality, this sum represents the total variability or sensitivity of the function $f(Z)$ with respect to changes in each individual component. It is a measure of how much the output of the function can vary due to changes in any single input variable.

Besides the splitting feature, there's a need to find the split value, which in a continuous attribute is more computationally demanding, with the most trending algorithm being the Gaussian Splitter. VFDT initially didn't support continuous numeric attributes until it was extended by the original authors in VFDTc, implementing a binary tree for each leaf node. For Categorical attributes, simpler statistical comparisons are sufficient and much less demanding. To find the split category, we can do a split for each one, group them by similarity criteria, or perform a one vs rest split. The latter can increase the number of potential splits exponentially.

## 4.2 Concept drift / Dynamic Architecture

Concept drift is the change over time in the probability distributions that govern the observed data, including the appearance of new features and the absence of initial ones. Different categories of concept drift, such as real and virtual drift, are described in depth in [31].

Adopting the division of techniques in [31] used to overcome concept drift, we picked the two that aligned with the research:

- Adaptive base learners

- Learners who modify the training data

The first one regards methods like the VFDT that use Hoeffding Bound to bring statistical guarantees that a split is valuable based on a confidence interval or EFDT, which adapts to concept drift by reevaluating splits and analyzing if the feature used to perform that split is still the best-performing. What both have in common is that their detection of concept drift is inherent, and not implying more complex computations. The second category relates to window methods and weighting methods. Window methods started with a fixed window of examples that defines the current concept for the node (CVFDT), which was later optimized to adaptive windows in each node, enhancing efficiency in computation and memory and requiring less user input (HAT). Weighting methods were implemented in ODTs in Option Trees and Fuzzy Trees, which implement a weighted voting technique since an example can reach various leaves, and each one contributes with a percentage to the final prediction.

Table 4.1: Algorithm's Concept Drift Detection

| Algorithms | Not Resistent to Concept Drift | Resistent to Concept Drift | |
|---|---|---|---|
| | | Indirectly | Window Method |
| VFDT | X | | |
| CVFDT | | | |
| HOT | | | X |
| HAT | | X | |
| EFDT | | X | |
| SVFDT | | X | |
| GAHT | | X | |
| EFHAT | | | X |
| M-VFDT | X | | |
| OSM | X | | |
| E-VFDT | X | | |
| IMAC | X | | |
| OVFDT | X | | |
| FEAT | | | X |
| Fuzzy | | | X |

## 4.3 Leaf Classification Strategy

In decision trees, leaves play a crucial role in classifying examples that reach them. When a leaf hasn't split yet, it still classifies the incoming data points, with one of the most effective techniques being Naive Bayes Adaptive method, which combines Majority Vote (MV) and Naive Bayes (NB) classifiers [7, 21, 22].

The key difference between these two classifiers lies in their memory requirements and computational complexity. MV operates by keeping a simple count of the number of examples for each class, which makes it extremely memory-efficient. It only needs to store the frequency of each class within the leaf,

requiring minimal storage and computational resources. In contrast, NB classifier needs to maintain a more detailed statistical representation of the data, including the probabilities of each feature given a class by storing the mean and variance for continuous features or frequency counts for categorical features for each class. Consequently, the NB classifier demands significantly more memory and computational power than MV.

Despite these differences, in the long run, the NB classifier is expected to converge towards the performance of the MV classifier [7]. As the number of examples increases, the probabilities calculated by NB will become more accurate, and the decision boundaries it forms will resemble those formed by the MV. Essentially, NB utilizes the additional statistical information to refine its predictions, but with a sufficiently large dataset, the difference between using NB and MV diminishes because the sheer volume of data will lead to a stable estimation of the underlying class distributions.

While a leaf hasn't split yet, it classifies examples reaching it, being the most frequent technique NB Adaptive, combining MV and NB. While the MV works by choosing the most frequent class, NB updates its statistics/probabilities as new data arrives, maintaining accuracy even as the data evolves. New examples in data can carry a different data distribution than the ones seen just before, especially in concept drift scenarios. That is why keeping a window of the latest examples or implementing weighting strategies prioritizing more recent examples can boost the prediction capabilities of the leaves. On the other hand, if the stream is static and little concept adapting will be needed, storing the Naive-Bayes statistics will have a negative impact on memory. We could implement Naive-Bayes as a cold-start technique to improve early accuracy, and when it stabilizes we discard Naive-Bayes metrics and only use MV labels counter.

## 4.4 Attributes

When considering the types of data that Online Decision Trees (ODTs) can process, attributes can be classified along three key dimensions: ordered or unordered, continuous or discrete, and ordinal or nominal [16, 35]. Among these dimensions, the distinction between continuous and discrete attributes has the most significant influence on computational demand, primarily because continuous attributes require more sophisticated methods to determine optimal splitting values.

For discrete attributes, the process of finding splits is relatively straightforward. The algorithm evaluates each unique value of the attribute as a potential threshold, calculates a splitting criterion (such as information gain or Gini impurity) for each potential split, and selects the one that optimizes the chosen criterion. This simplicity results in lower computational demand compared to continuous attributes.

Splitting continuous attributes is complex due to the infinite number of potential split points. In VFDTc, this involves maintaining a binary search tree at each leaf node, dynamically inserting new values, and tracking class counts on either side of each cut point [22]. Information gain is calculated for each potential cut point, and the optimal one, which minimizes entropy, is selected to ensure efficient decision tree

learning. More advanced techniques like the Gaussian splitter eventually replaced this initial method. This splitter assumes data follows a Gaussian distribution, estimating each class's mean and standard deviation allowing for more accurate split points by leveraging the statistical properties of the data, resulting in better handling of continuous attributes and improved overall performance.

Memory usage differs significantly between discrete and continuous attributes due to the nature of the statistics that need to be maintained. For discrete attributes, memory requirements are relatively low because the algorithm only needs to store the frequency counts of each unique value [11]. In contrast, continuous attributes demand more memory. The binary search tree approach used in VFDTc requires storing all observed values and their associated counts, which can grow large over time. In comparison, the Gaussian splitter requires less memory, as it only needs to store and update the mean and standard deviation for each class. However, while the Gaussian splitter is more memory-efficient, it is more demanding in processing power due to the additional computations required for estimating and updating the statistical parameters [50].

# Chapter 5

# Optimizations

This chapter will delve into further optimizations focusing on splitting rules and concept-drift adaptation. Ensuring that ODTs are adaptive is crucial for maintaining their performance in dynamic environments. Therefore, the techniques discussed in this chapter aim to enhance Incremental Learning, enabling ODTs to process data streams and adapt to changes over time efficiently. The structure of this chapter is presented in Figure 5.1, which outlines the key areas of optimization.

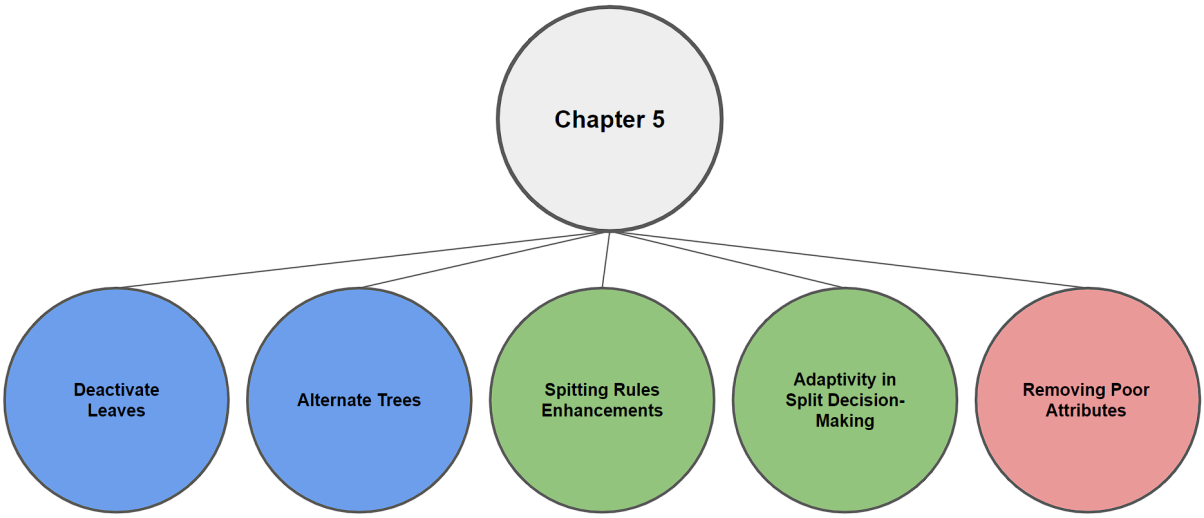


Figure 5.1: Chapter 5 Structure

## 5.1 Deactivate Leaves

In the base implementation of the HT, VFDT, when maximum memory is reached, or leaf nodes barely see new examples, leaves are deactivated, saving their statistics and replacing them with a new leaf [17]. If a deactivated leaf outperforms a currently activated leaf, it can be replaced again to improve

the tree's accuracy. We provide a visual explanation in figure 5.2. The first nodes are deactivated, but their statistics are saved. The node can perform a split again when sufficient statistics are stored, but those new leaves can be replaced with the deactivated ones if they are outperformed. While VFDT only evaluates leaf nodes, EFDT re-evaluates every node in the tree to ensure that the current split is the best. If an improved split is detected, a new leaf node will replace the whole subtree rooted at the faulty node. The EFDT can drastically reduce variance by simplifying the model, yet it can result in significant bias if important information is lost by dropping the subtree [42].

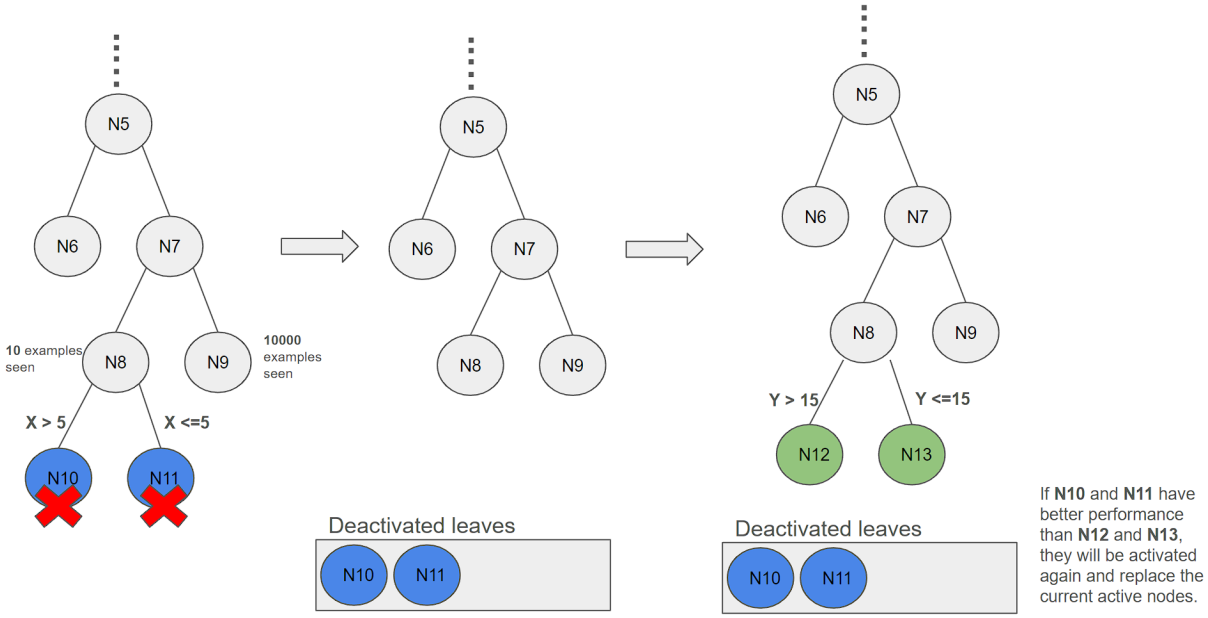


Figure 5.2: Deactive Leaves Mechanism

## 5.2 Alternate Trees

From the algorithms gathered, three adapt to concept drifting streams while training temporary trees to compare to a node's subtree where concept drift was detected. This alternate tree replaces the subtree rooted at the faulty node when its accuracy is superior and after a user-defined number of examples. In figure 5.3, we have a vision representation of this mechanism, where ADWIN detects a change and begins training an alternate tree that replaces the original one. This mechanism reduces bias while keeping the model relevant, ensuring the new subtree is adapted to the new concept [33, 4, 41].

The first implementation was CVFDT, followed by HAT and, ultimately, EFHAT, which combines HAT and EFDT. We can observe significant enhancements, each time improving the tree's performance. They all use window methods to detect error changes in prediction. CVFDT uses a fixed window size, which limits its performance since, when the stream is stable, a smaller window wouldn't affect algorithm accuracy but would significantly increase its computing performance. HAT introduces ADWIN as a

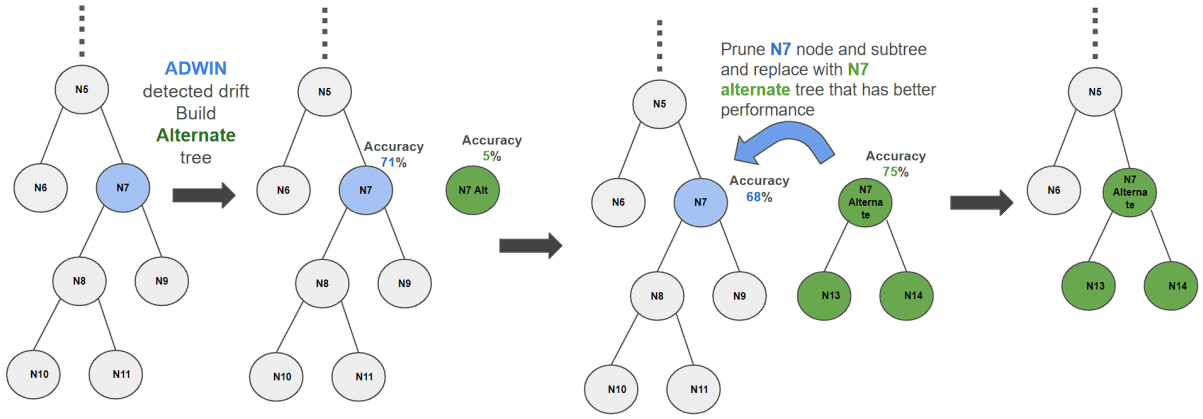


Figure 5.3: Alternate Tree Mechanism

change detector, an adaptive window method that expands or diminishes the window of examples of each node based on the current concept. EFHAT is an extension of HAT but implementing the EFDT splitting rule, meaning that splits happen when splitting on the best attribute is better than not splitting with required confidence using the Hoeffding Bound.

While CVFDT requires that another splitting attribute becomes the top split attribute, HAT and EFHAT start growing a new alternate subtree based on change detection possibly rooted with the same split attribute as the subtree being replaced.

## 5.3 Splitting Rules Enhancements

The standard Hoeffding Tree decision tree evaluates the top attributes and compares if the best attribute is superior to the second by a confidence interval defined by the Hoeffding Bound. Besides the Hoeffding Bound guarantees, some implementations have added extra splitting rules to promote even more "valuable" splits [24, 15, 32, 60].

In Tie Breaking in Hoeffding Trees the authors introduced Genuine Tie Detection (htdt), which validates that the best splitting attributes are indeed superior than the rest. It compares the difference between the best and second-best candidates with the difference between the second-best and worst candidates, where the second value is expected to be significantly superior.

SVFDT implemented two groups of rules, one leading to a faster expansion and another to better accuracy. These rules compare leaf metrics like examples seen, such as Entropy and Information Gain, against the average of all other leaves. In most of these rules, the statistics of leaves taken into consideration are the ones saved each time a leaf fulfills VFDT splitting rules (Fulfilling VFDT split rules doesn't mean that splitting occurred in SVFDT since there are extra rules to consider). SVFDT splitting rules must be chosen before running the algorithm.

GAHT also offers two sets of splitting rules that alternate during algorithm execution. Those rules alternate based on a metric called fraction that represents a normalized measure of how many instances have been observed at a particular leaf relative to the average number of instances per leaf since the leaf was created. By having user-defined boundaries, we divide the fraction into three intervals that, in ascending order, apply the following to the leaf: Deactivate leaf, VFDT splitting rules, EFDT splitting rules. The higher the fraction metric, the more significant that leaf is and the more up-to-date with the current concept, so we relax splitting, reducing bias. The fraction metric works by first assessing the contribution of each leaf to the model's learning process, achieved by dividing the number of examples a leaf has seen by the total number of examples processed by the tree since that leaf's creation, gauging the relative importance of the leaf. To make this contribution comparable across different stages of the tree's growth and varying tree sizes, the ratio is further normalized by dividing it by the total number of leaves in the tree. This normalization ensures that the metric remains consistent and meaningful, regardless of how the tree evolves.

OVFDT measures accuracy through the difference between the number of wrongly classified samples and those correctly classified. It also keeps track of Hoeffding Bound fluctuation, the mean, minimum, and maximum values. They maintain the standard splitting rule for VFDT but attach three new rules that one needs to fulfill for the split to occur. Those rules include comparisons of the differences in Information Gain and accuracy metrics with Hoeffding Bound values (minimum and maximum), reducing variance by being more restrictive when performing splits.

The accuracy metric at an iteration is the difference between correctly classified samples (number of examples that fall into existent leaves) and incorrectly classified samples (number of examples that don't fall into existent leaves).

## 5.4 Adaptivity in Split Decision-Making

In decision tree expansion, a leaf always enforces the Grace Period, the number of examples a leaf needs to train on before attempting a split. If a split fails in the next evaluation, the gain must surpass the Tie Threshold to perform the split. These parameters are user-defined in the VFDT however, [24] showed that having the Grace Period dynamically calculated will reduce energy consumption and only have a minor effect on accuracy. Low energy consumption is a requisite for algorithms running on ubiquitous devices since most are battery-powered, and it benefits scalability by diminishing energy costs. Besides this upside, having dynamic parameters for splitting allows for faster growth of high-confidence branches. It delays the growth of less confident ones, improving the bias-variance trade-off, reducing bias by rapidly expanding some branches and reducing variance by slowly expanding others. Addressing the Adaptive Grace Period, the two algorithms that implemented it store data distribution statistics and calculate the Grace Period for the newly formed leaf. Regarding the Adaptive Tie Threshold, the first implementation, [32], had a more straightforward solution, which involved having an increased wait

period when a tie occurs. If a leaf reaches a tie-breaking situation, eventually, after splitting occurs, that leaf's child leaves will have the tie wait period increased by a fixed amount. The effect is cumulative, meaning that if a leaf reaches a tie, its child leaf will have an increased tie wait period by the fixed amount, and again, if this child reaches a tie situation, its child leaf will have the same wait period as the parent plus an increase. However, if that leaf can split without a tie event, the tie wait period is reset. In the other implementations [55, 37, 59], statistics about Hoeffding Bound fluctuation are saved to calculate the new tie threshold.

## 5.5 Removing Poor Attributes

The last optimization found in this study relates to the number of attributes tested in a split. As it was pointed out by [24], splitting numeric attributes is more demanding in computational power, so reducing the number of attributes tested will reduce the computing time of split calculation, even more significant if removed attributes are numeric. Algorithms that implemented this initially are E-VFDT, IMAC, and FEAT but we also found this feature in most VFDT implementations or extensions available in the various frameworks such as MOA [9] and River [43].

Table 5.1 organizes all the information discussed, identifying the optimizations that each algorithm focused on. The algorithms references can be found in table 3.2

Table 5.1: Algorithms and their Optimizations

| **Algorithms** | **Prune** | | **Extra Splitting rules** | **Adaptive Grace Period** | **Adaptive Tie Threshold** | **Removal of Poor Attributes (RPA)** |
|---|---|---|---|---|---|---|
| | **Deactivate Leaf** | **Alternate Trees** | | | | |
| VFDT | X | | | | | |
| CVFDT | | X | | | | |
| HOT | | | | | | |
| HAT | | X | | | | |
| EFDT | | | X | | | |
| SVFDT | | X | | | | |
| GAHT | | X | | X | | |
| EFHAT | | X | | | | |
| M-VFDT | | | | | X | |
| OSM | | | | X | | X |
| E-VFDT | | | | | X | |
| Tie Breaking in Hoeffding Trees | | | X | | X | |
| IMAC | | | | | X | |
| OVFDT | | | | | | X |
| FEAT | | | X | | | X |

# Chapter 6

# Experiments

This chapter presents a comprehensive analysis of our experimental setup and methodology to evaluate the performance of various machine learning algorithms for data streams. We aim to assess the effectiveness of different algorithmic hyper-parameters and compare the performance of individual models and ensemble methods using synthetic and real-world datasets.

We start by detailing the tools and libraries employed in our experiments, such as MOA, River, and the newly released CapyMOA, all designed for machine learning on data streams. Following this, we provide an overview of the datasets used. Synthetic datasets create controlled environments with adjustable concept drifts, enabling an in-depth analysis of algorithm behavior under varying conditions. Conversely, real-world datasets are benchmarks for evaluating algorithm performance in practical applications.

The algorithms tested in our experiments include the Hoeffding Tree (HT), Extremely Fast Decision Tree (EFDT), Hoeffding Adaptive Tree (HAT), Leverage Bagging (LBag), Adaptive Random Forest (ARF), and Streaming Random Patches (SRP). Our experimental design, illustrated in figure 6.2, outlines a systematic approach to ensure a thorough and unbiased evaluation of the algorithms across different datasets and conditions.

## 6.1 Experimental Setup

From the several algorithms presented in chapter 3, most authors provide the code and identify what framework it was built on so it can be tested. Implementation languages vary between Java, C++ (Cython), and Python.

A frequently adopted framework by researchers is Massive-Online-Analysis (MOA) [9]. It is implemented in Java and provides a collection of machine learning algorithms for various use cases (classification, regression, outlier or concept drift detection, recommender systems, and clustering) and tools for

evaluation and datasets to experiment on. In May of 2024, the team released a new framework, CapyMOA, that extends MOA and is tailored to be faster and more efficient. Through a unified streaming API, it integrates libraries such as scikit-learn and Pytorch, allowing programming to be done in Java or Python.

Another prominent framework in the realm of stream learning is River [43]. River is a Python package designed for dynamic data streams and continual learning. It merges the capabilities of two prior libraries, Creme and scikit-multiflow, to offer an advanced framework for stream learning applications.

Integrating CapyMOA, MOA, and River in our experimental setup allows us to leverage the strengths of each framework and not be biased toward one setup. Other data-streaming frameworks are:

- Apache SAMOA
- streamDM C++
- ADAMS
- MEKA

### 6.1.1 Datasets

The datasets used in our experiments include both synthetic and real-world data. Synthetic datasets, such as the Agrawal, LEDDrift, SEA, HyperPlane, and RBFDrift, are generated to simulate controlled environments, some with tunable concept drifts, allowing for a detailed analysis of algorithm behavior under various scenarios. Real-world datasets, like the Sensors dataset from CapyMOA, provide a benchmark for assessing the algorithms' performance in practical applications. The choice of datasets ensures a comprehensive evaluation of the algorithms across different data types and drift scenarios.

Figure 6.1 illustrates synthetic datasets used in experiments with the River and CapyMOA frameworks, showcasing the introduction of concept drifts. In the CapyMOA Synthetic dataset, which comprises 1,000,000 instances, three concept drifts are introduced: the first drift is gradual, followed by an abrupt drift, and concluding with another gradual drift. Conversely, the River Synthetic dataset, limited to 500,000 instances, features two concept drifts: an initial gradual drift followed by an abrupt drift.

### 6.1.2 Evaluation

The evaluation procedure of a learning algorithm determines which examples are used for training the algorithm and which are used to test the model output by the algorithm. Historically, in batch learning, the choice of evaluation procedure has partly depended on data size. As data sizes increase, practical time limitations prevent procedures that repeat training too many times. With considerably larger data sources, reducing the number of repetitions or folds is commonly accepted to allow experiments to be completed in a reasonable time.

Two main approaches arise :

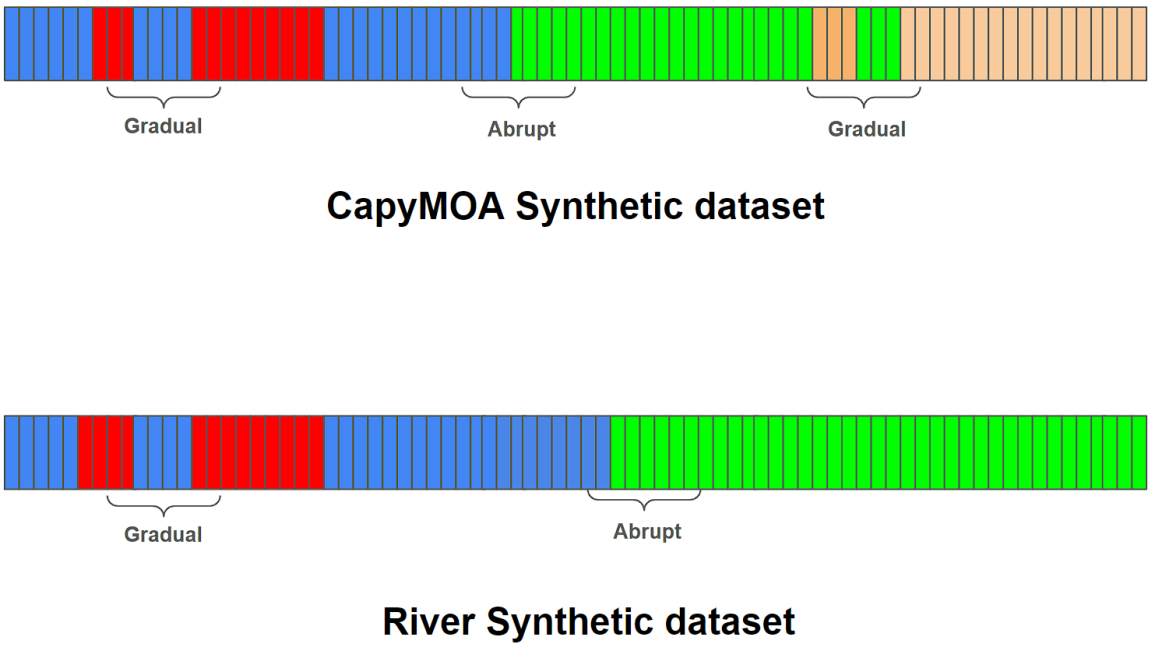


Figure 6.1: Synthetic datasets with custom drift for MOA and River

- Holdout
- Interleaved Test-Then-Train (or Prequential)

The Holdout approach is often used in traditional batch learning when scale makes cross-validation too time-consuming. This method measures performance on a single holdout set, particularly useful when the division between train and test sets has been pre-defined, enabling direct comparisons across different studies. On the other hand, the Interleaved Test-Then-Train approach involves using each example to test the model before using it for training, thereby incrementally updating the accuracy. This scheme ensures that the model is always tested on unseen examples, negating the need for a separate holdout set and maximizing the available data. Additionally, it provides a smooth plot of accuracy over time, as each example's significance to the overall average decreases incrementally.

To evaluate the performance of decision trees, we consider two groups of metrics. The first group relates to memory usage, which can be measured by the bytes occupied and visualized with tree structure descriptions such as the number of leaves, height, and number of branches. In algorithms using alternate trees or inactive leaves, their impact on memory can also be evaluated. These characteristics influence the memory footprint of the tree, and tree expansion can be influenced by tuning hyperparameters like the grace period and tie threshold.

The second group of metrics relates to performance, with accuracy being a typical measure. However, more valuable metrics are often preferred in data streams due to the unique challenges presented by this type of data. For instance, AUC (Area Under the Curve) is a better metric for comparing performance

as it is robust to class imbalances, making it more reliable when dealing with skewed datasets. We will also employ the F1-Measure, which balances precision and recall, giving a more comprehensive view of the classifier's performance. Precision measures the accuracy of the positive predictions, and recall measures the model's ability to capture all relevant instances.

In some studies, such as [23], Garcia profiled the energy consumption of VFDT to identify algorithmic hotspots. For edge devices with limited memory, like those used in TinyML applications, the NetScore metric, as utilized by [29], is invaluable for assessing and optimizing the trade-offs between accuracy, memory usage, and computational speed. This metric promotes a balanced evaluation by combining different metrics, helping to prioritize models that achieve acceptable accuracy while minimizing memory and computational demands. Another valuable metric in resource-constrained environments is RAM-Hours [8], which provides a comprehensive view of an algorithm's memory efficiency over time. RAM-Hours combines the amount of RAM used and the duration of usage, allowing researchers to identify algorithms that maintain an optimal balance between accuracy and resource consumption.

### 6.1.3 Setup

The following figure 6.2 describes the experiments done describing the framework, experiment, instances, dataset and trees tested.

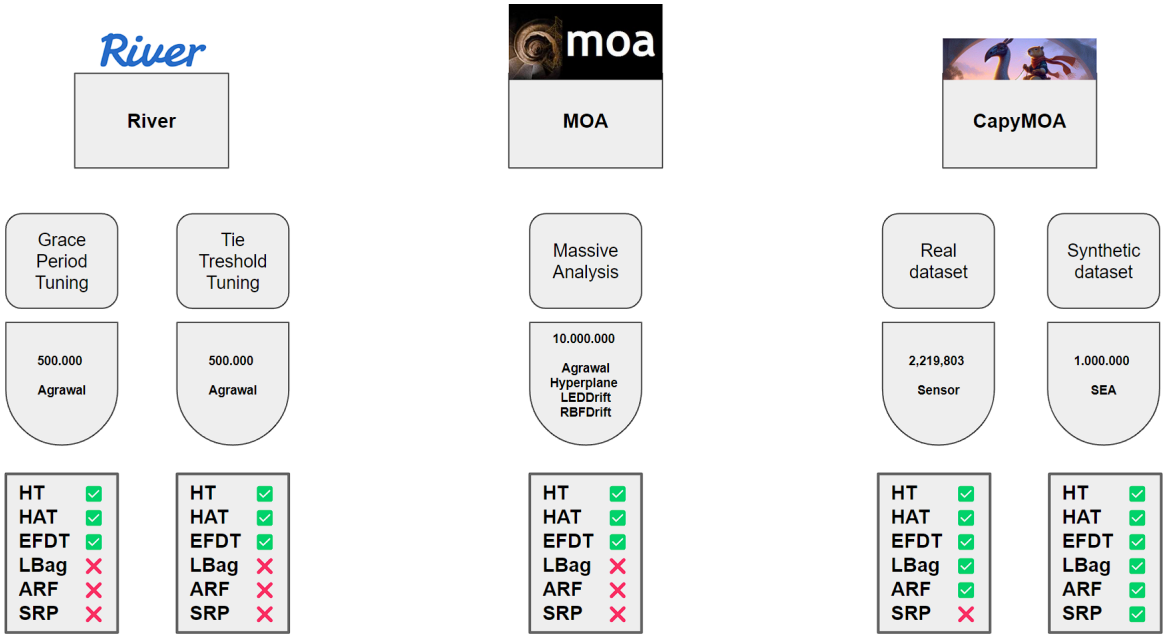


Figure 6.2: Experimental Setup

Each framework was selected for specific tasks based on its strengths and limitations. River, implemented in Python, facilitated easy extraction and combination of metrics into plots, making it ideal for analyzing Grace Period and Tie Threshold tuning despite its slower runtime compared to MOA. River's

Python implementation allowed to analyze many more metrics either relating to performance or tree structure. We ran these experiments on Agrawal synthetic dataset with 500000 instances implementing drift at 150000(incremental) and 300000 (abrupt).

MOA and CapyMOA offered comparable run speeds, with CapyMOA being slightly faster. However, MOA's Java implementation presented challenges in running and modifying code, so we used MOA primarily for straightforward evaluations. Specifically, we compared the performance of single trees (HT, HAT, and EFDT) across four different datasets, each with 10 million instances. CapyMOA, as the latest implementation with updated versions of ensembles, was used to compare single trees and ensemble performance. This included testing on a real-world dataset, Sensors, and a custom synthetic dataset generated from SEA with two concept drifts. In the synthetic dataset, we tested three ensembles: LBag, ARF, and SRP. However, in the real-world dataset, we could only finish training LBag and ARF due to memory constraints with SRP.

## 6.2 River

### 6.2.1 Grace Period

The objective of this experiment is to investigate the impact of tuning the grace period parameter on the performance of three decision tree algorithms: Hoeffding Tree (HT), Extremely Fast Decision Tree (EFDT), and Hoeffding Adaptive Tree (HAT). The grace period plays a vital role in the functioning of decision trees, particularly in the context of streaming data. A shorter grace period allows the tree to split nodes more frequently, potentially leading to quicker adaptation to changes in the data stream. However, this can also result in higher computational overhead and the risk of overfitting. Conversely, a longer grace period reduces the frequency of splits, which may enhance computational efficiency and generalization but can delay the model's responsiveness to significant changes in the data stream. Therefore, tuning the grace period is essential for optimizing the performance of decision trees in dynamic environments.

The Hoeffding Tree (HT) models in Figure 6.3 show distinct differences across the grace periods. The GP10_HT configuration achieves rapid initial accuracy, peaking around 0.85, but it suffers more from both the incremental drift at 150k instances and the abrupt drift at 300k instances, showing significant dips before recovering slightly. It also has the longest running time, steadily increasing due to frequent updates, and the highest, most volatile memory usage, reflecting frequent model adjustments and higher complexity with more nodes, branches, and leaves. In contrast, GP200_HT and GP1000_HT configurations show smoother accuracy curves, better stability post-drifts, and achieve a higher final accuracy close to 0.87. They have more controlled increases in running time and lower, more stable memory usage, indicating fewer, more substantial updates that enhance efficiency and reduce complexity. Overall, GP1000_HT offers the best balance of accuracy, stability, and resource efficiency, making it the most effective configuration for handling both incremental and abrupt drifts.

Table 6.1: Metrics used by different frameworks

| Framework | Metrics |
|---|---|
| **River** | • **Accuracy**<br>– BalancedAccuracy<br>• **Resource Usage**<br>– Time<br>– memory<br>– RAM-Hours<br>• **Tree Structure**<br>– N. nodes (number of nodes)<br>– N. branches (number of branches)<br>– N. leaves (number of leaves)<br>– height |
| **MOA** | • **Accuracy**<br>– Classifications Correct (percent)<br>– Kappa Statistic (percent)<br>• **Resource Usage**<br>– Evaluation Time (CPU seconds)<br>– Model Cost (RAM-Hours)<br>– Model Serialized Size (bytes)<br>• **Tree Structure**<br>– Tree Size (nodes)<br>– Tree Size (leaves)<br>– Active Learning Leaves<br>– Tree Depth |
| **CapyMOA** | • **Accuracy**<br>– Classifications Correct (percent)<br>– Kappa Statistic (percent)<br>– F1 Score (percent)<br>– Precision (percent)<br>– Recall (percent) |

The HAT (Hoeffding Adaptive Tree) algorithm's performance with different grace periods (GP), in Figure 6.4, shows a clear trade-off between accuracy and computational resources. Lower grace periods like GP_10 achieve higher balanced accuracy but at the cost of increased time, memory usage, and computational cost (RAM hours). As the number of instances grows, GP10_HAT also results in the most nodes, branches, and leaves, leading to a more complex and resource-intensive tree. Conversely, higher grace periods like GP_1000 reduce these resource demands but result in lower accuracy. GP200_HAT falls between the two, balancing accuracy and resource usage moderately. Therefore, selecting an ap-

propriate grace period depends on the specific balance needed between model accuracy and available computational resources.

The Extremely Fast Decision Tree (EFDT) models 6.5 exhibit the most pronounced impact of grace periods on performance and resource usage. The GP10_EFHT configuration achieves high initial accuracy, reaching about 0.84, but shows significant drops at both the incremental drift at 150k and the abrupt drift at 300k instances, with high and fluctuating running times and memory usage, indicating intensive computational demands and frequent model adjustments. GP200_EFHT and GP1000_EFHT configurations, however, offer improved stability and better handling of concept drifts, with a final accuracy of around 0.87. These models benefit from lower and more stable running times and memory usage, demonstrating efficient resource management. GP1000_EFHT shows the most stable trend regarding balanced accuracy and the least computational cost, making it the best configuration for balancing accuracy and resource efficiency while effectively managing both concepts drifts.

### 6.2.2 Tie Treshold

The objective of this experiment is to examine the effect of tuning the tie threshold parameter on the performance of the Hoeffding Tree (HT), Extremely Fast Decision Tree (EFDT), and Hoeffding Adaptive Tree (HAT) algorithms. The tie threshold is crucial in decision tree splits as it helps in making more confident and reliable splitting decisions. A lower tie threshold allows splits to occur even when there is only a small difference between the best and second-best split options, which can lead to more frequent but potentially less reliable splits. This may enhance the tree's responsiveness to changes in the data stream but could also increase the risk of overfitting and computational burden. On the other hand, a higher tie threshold requires a more substantial difference to split, promoting more conservative and potentially more accurate decision-making at the cost of slower adaptation. Therefore, tuning the tie threshold is vital for balancing the trade-offs between split reliability, computational efficiency, and the tree's ability to adapt to evolving data streams.

In Hoeffding Tree (HT) models, in Figure 6.6, varying the tie threshold significantly affects performance metrics and model complexity. The Tie0.005_HT configuration shows a steady initial increase in accuracy but struggles to handle both the incremental drift at 150k and the abrupt drift at 300k instances, ultimately stabilizing around 0.75. This configuration has the lowest and most stable memory usage due to fewer updates, reflected in its lower complexity with fewer nodes, branches, and leaves. Conversely, the Tie0.05_HT configuration performs better, managing drifts more effectively and achieving a final accuracy close to 0.85, with moderate memory usage and complexity. The Tie5_HT configuration excels by achieving the highest initial and final accuracy, around 0.9, and maintaining stability through the drift points. This configuration, however, incurs the highest memory usage and complexity due to frequent updates required for maintaining high accuracy. In summary, Tie05_HT stands out as the best configuration since it balances accuracy and resource consumption.

Hoeffding Adaptive Tree (HAT) models, in Figure 6.7, benefit significantly from higher tie thresholds.

The Tie0.005_HAT configuration starts with lower initial accuracy, around 0.7, and similar to HT experiment it struggles with both incremental and abrupt drifts, resulting in a lower final accuracy compared to other configurations. It has the lowest and most stable memory usage and complexity due to infrequent updates. The Tie0.05_HAT configuration performs better, handling drifts more effectively and reaching a final accuracy of around 0.85, with moderate memory usage and complexity. The Tie5_HAT configuration achieves the highest initial and final accuracy, around 0.95, and maintains stability through both drift points. This configuration incurs the highest memory usage and complexity, reflecting the frequent updates required for high accuracy. Overall, Tie5_HAT provides the highest accuracy and stability, and when compared to the HT experiment, it doesn't consume as much memory because the tree concept drift detection and adaptation prunes the tree significantly when it acts. With further enhancement, we could optimize HAT to use a less strict tie threshold and promote expansion but prune more frequently so as not to demand too much memory.

Like in the other models, with EFDT models, in Figure 6.8, the Tie0.005_EFHT configuration shows slower initial improvement and significant dips at both drift points, stabilizing around 0.75, with lower and more stable memory usage and complexity due to infrequent updates. The Tie0.05_EFHT configuration performs better, handling drifts more effectively and reaching a final accuracy of around 0.85, with moderate memory usage and complexity. The Tie5_EFHT configuration achieves the highest accuracy, around 0.9, with good stability through drift points. However, it incurs the highest running time and memory usage due to frequent computations triggered by the high tie threshold before splits occur.

## 6.3 MOA

With MOA framework we test the single trees - HT, HAT and EFDT - across four different datasets with 10 million instances. Two of them have concept drift while the other two are stable. The stables ones are Agrawal and Hyperplane and the ones with drift are RBFDrift and LEDDrift. The collection of all metrics can be observed in Figures 6.13 and 6.14.

The Agrawal dataset shows in Figure 6.9 all three algorithms achieving high accuracy, around 95%, with EFDT and HAT slightly outperforming HT. Memory usage for HT is the highest, showing significant spikes, while HAT and EFDT have lower and more stable memory consumption. EFDT is the most efficient in terms of runtime, with HT taking the longest and HAT being moderate. EFDT's superior handling of both incremental and abrupt drifts is highlighted by its consistent performance and resource efficiency.

In the HyperPlane dataset, in Figure 6.10, all three algorithms—HT, HAT, and EFDT—demonstrate comparable accuracy, fluctuating between 88% and 92%. EFDT shows a slight edge in stability. HAT consumes the most memory, which increases significantly over time, while HT has moderate memory usage. EFDT is the most efficient in terms of memory consumption. When it comes to runtime, HT takes the longest, followed by HAT, with EFDT being the most time-efficient. EFDT's efficiency can be

attributed to its ability to handle splits more effectively, which reduces computational overhead.

For the LED Drift dataset, in Figure 6.11, the algorithms again show similar accuracy, fluctuating between 70% and 76%. EFDT has a slight advantage in maintaining higher accuracy. HAT uses the most memory, which increases steadily throughout the process, while HT and EFDT have more moderate and stable memory usage. Runtime analysis shows HT as the slowest, HAT as moderate, and EFDT as the fastest. EFDT's efficient handling of splits contributes to its lower runtime and more stable memory usage, making it well-suited for datasets with gradual drifts.

In the RBF Drift dataset, in Figure 6.12, all three algorithms maintain high accuracy around 95% after initial fluctuations, with HAT achieving the highest peak accuracy. However, HAT also consumes the most memory, which grows significantly over time, followed by HT and then EFDT. Runtime analysis shows HT taking the longest time, HAT as moderate, and EFDT being the most efficient. The EFDT algorithm's ability to manage abrupt drifts efficiently is evident in its balanced performance across accuracy, memory usage, and runtime.

.

## 6.4 CapyMOA

### 6.4.1 Real World dataset

In this experiment we used Sensors real-world dataset available in CapyMOA to compare HT, HAT, EFDT, LBag and ARF. The multiple plots covering accuracy, memory and training time. 6.15.

The analysis of classification algorithms on the given dataset reveals that ARF (Adaptive Random Forest) and LBag (Leveraging Bagging) outperform other models across various performance metrics. Both ARF and LBag consistently achieve high classification accuracy, maintaining around 90%, and show robust Kappa Statistics frequently exceeding 80%, indicating strong agreement beyond chance. These algorithms also excel in precision and recall, both averaging above 80%, which reflects their ability to make accurate and comprehensive positive predictions. The F1 Scores for ARF and LBag, often above 80%, further underscore their balanced performance, making them reliable and effective choices for classification tasks.

In contrast, EFDT (Extremely Fast Decision Tree) demonstrates moderate performance, particularly excelling in the Kappa Statistic and maintaining reasonable precision and recall rates, though it does not match the levels of ARF and LBag. HAT (Hoeffding Adaptive Tree) and HT (Hoeffding Tree) exhibit significantly poorer performance, with high variability and lower values in classification accuracy, Kappa Statistic, precision, recall, and F1 Score. These metrics often fall below 60% for accuracy and 50% for Kappa, indicating less reliable and effective performance. Consequently, ARF and LBag stand out as the preferred algorithms for this dataset, providing consistently superior results across all evaluated metrics.

Table 6.2: Performance metrics for various algorithms

| Algorithm | F1 Score (%) | Kappa Statistic (%) | Classifications Correct (%) | Precision (%) | Recall (%) |
|---|---|---|---|---|---|
| HT | 46.12 | 42.18 | 44.50 | 30.23 | 43.38 |
| HAT | 50.67 | 42.54 | 45.21 | 60.53 | 48.95 |
| EFDT | 79.30 | 77.52 | 78.57 | 57.58 | 76.95 |
| LBag | 83.60 | 83.69 | 84.49 | 82.30 | 82.35 |
| ARF | **86.87** | **87.04** | **87.68** | **85.69** | **85.46** |

### 6.4.2 Synthetic

In this experiment we used the SEA dataset and implemented custom drifts to observe algorithms behaviour. We generated 1 million instances and applied three drifts, as can be observed in red in the images (two gradual drifts and one abrupt).

Overall we can see in 6.16, HoeffdingAdaptiveTree (HAT) is the most resilient and adaptable algorithm among the three, consistently maintaining high performance and demonstrating quick recovery during both gradual and abrupt concept drifts. HoeffdingTree (HT) and Extremely Fast Decision Tree (EFDT) both struggle more during concept drifts, particularly the second gradual drift, with HT showing the most significant challenges in recovery and stability. EFDT performs slightly better than HT in terms of recovery but is still less robust than HAT. These observations highlight HAT as the preferred choice for environments with frequent and varied concept changes, due to its superior adaptability and resilience.

In comparing the performance of Hoeffding trees with ensemble methods 6.17, it becomes evident that StreamingRandomPatches (SRP) didn't outperform HoeffdingAdaptiveTree (HAT) and other ensemble methods. SRP fails to recover after the abrupt drift and continues to degrade during the second gradual drift, indicating poor adaptability to dynamic environments. On the other hand, AdaptiveRandomForest (ARF) demonstrates resilience comparable to HAT, maintaining high performance during gradual drifts and recovering well after the abrupt drift. This strong adaptability makes ARF one of the most robust choices among both single-tree and ensemble methods. Leveraging Bagging (LBag) performs slightly better than SRP and is on par with ARF in terms of handling concept drifts, showing quick recovery after the abrupt drift and maintaining high accuracy during gradual drifts, similar to HAT. Overall, among the ensemble methods, ARF and LB exhibit robust performance and adaptability to both abrupt and gradual concept drifts, making them reliable alternatives to HAT. In contrast, SRP struggles significantly with concept drifts, especially the second gradual drift, failing to recover effectively and proving less suitable for dynamic environments.

## 6.5 Summary and Conclusion

Our experiments demonstrate that a lower grace period significantly enhances performance and adaptability post-drift. This improvement is attributed to the increased frequency of split attempts, which, although resource-intensive, facilitates rapid adaptation. Implementing an adaptive grace period strategy—employing a higher interval during stable phases and a lower interval during concept drift—would promote faster adaptation and optimize resource usage.

In comparing Hoeffding Adaptive Tree (HAT) with Hoeffding Tree (HT), we find that HAT is more resource-efficient due to its ability to detect changes and aggressively prune the tree, unlike HT, where the tree size steadily increases.

The tie threshold experiment revealed that a tau value of 0.005 is too low for meaningful results, as all HT implementations fail to grow, resulting in poor accuracy and minimal memory usage. Higher thresholds promote tree expansion, leading to larger structures and improved accuracy. While HT trees grow rapidly with high thresholds, HAT manages expansion conservatively through pruning.

From the first experiment, it is evident that hyperparameters significantly influence tree expansion and metrics. Dynamic hyperparameters are valuable for promoting rapid adaptation and growth when necessary, while stabilizing expansion during satisfactory performance. Frequent expansions with high tie thresholds or low grace periods are less problematic in HAT and EFDT compared to HT, where tree expansion is a major issue.

Analyzing the behavior of different algorithms using the MOA framework across various datasets, we observe distinct memory usage patterns. For instance, HT occupies more memory than HAT in the Agrawal dataset, while the opposite is true for the RBFDrift dataset. The LEDDrift dataset, with its numerous noisy features, results in smaller tree sizes despite lower performance, indicating noise recognition. Complex datasets like RandomRBF and HyperPlane lead to models with high memory usage, with RBFDrift creating deeper trees, resulting in longer training times but superior accuracy by 5

Comparing single HT implementations with state-of-the-art ensembles using CapyMOA, we find that HT is consistently outperformed by more recent algorithms, particularly in real-world datasets. HT shows the lowest performance, followed by HAT and EFDT. The significant performance degradation towards the end suggests a major concept drift, with HAT adapting by pruning its tree, while HT fails to detect the drift. EFDT's dynamic structure mitigates the impact of concept drift better than HT.

Ensemble methods exhibit robust performance, maintaining stability with minimal performance decline during concept drifts. Although further experiments are required to explore tree architecture and memory usage in detail, CapyMOA's current implementation lacks these metrics. Altering the code to obtain these metrics would compromise CapyMOA's execution speed, and thus, this aspect is reserved for future research.

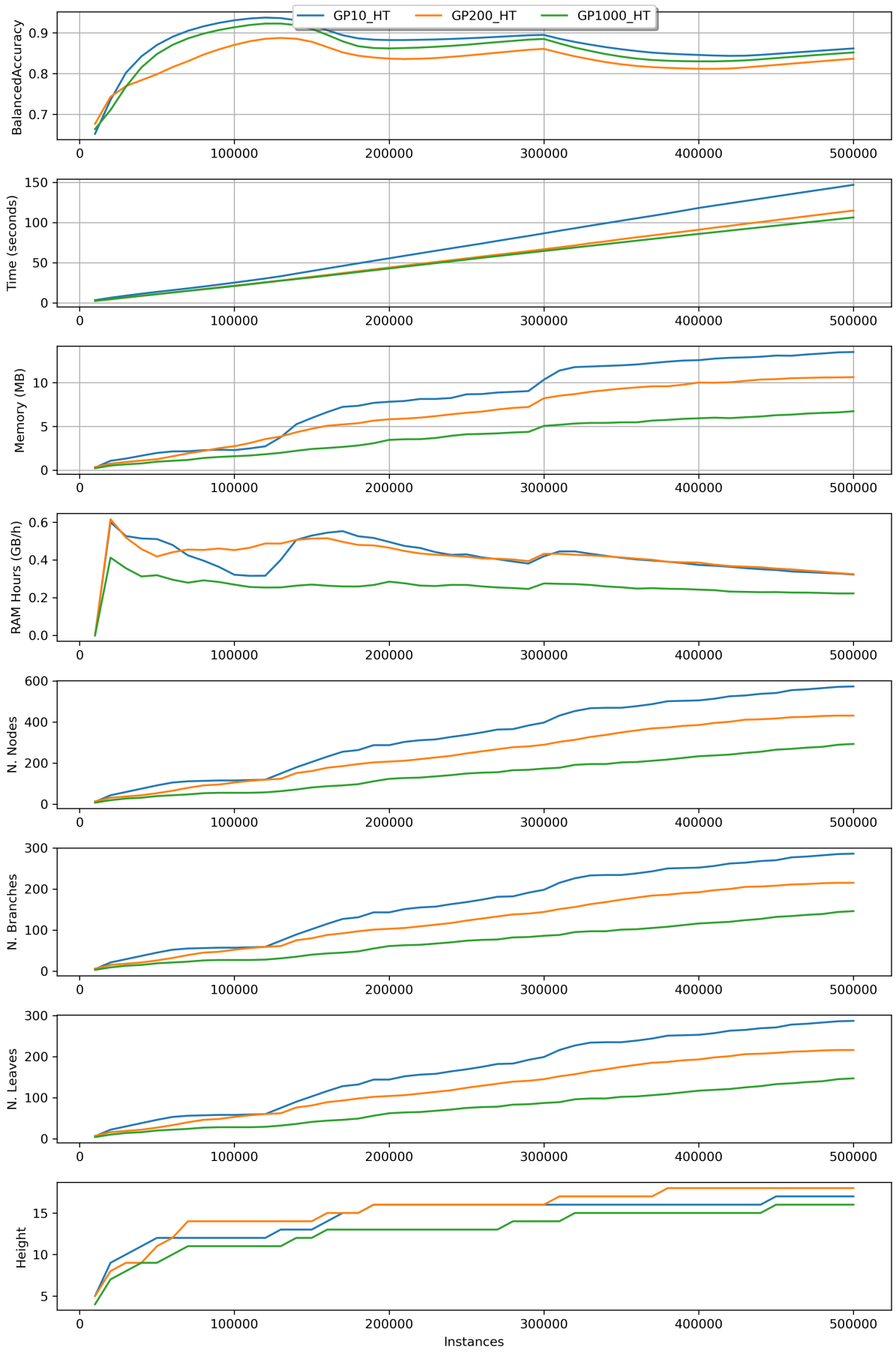


Figure 6.3: (River - Grace Period) Agrawal with 1M instances - HT

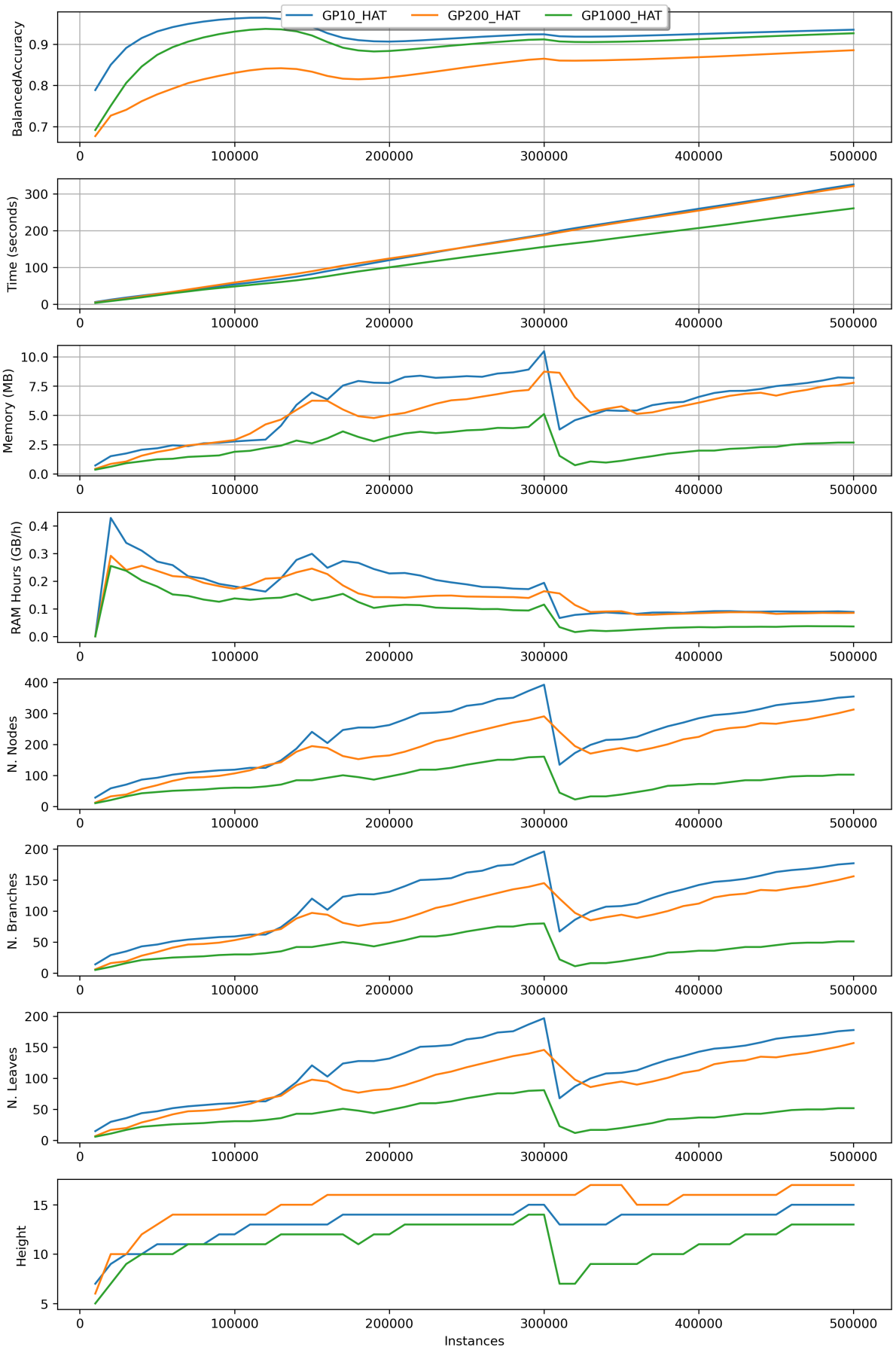


Figure 6.4: (River - Grace Period) Agrawal with 500K instances - HAT

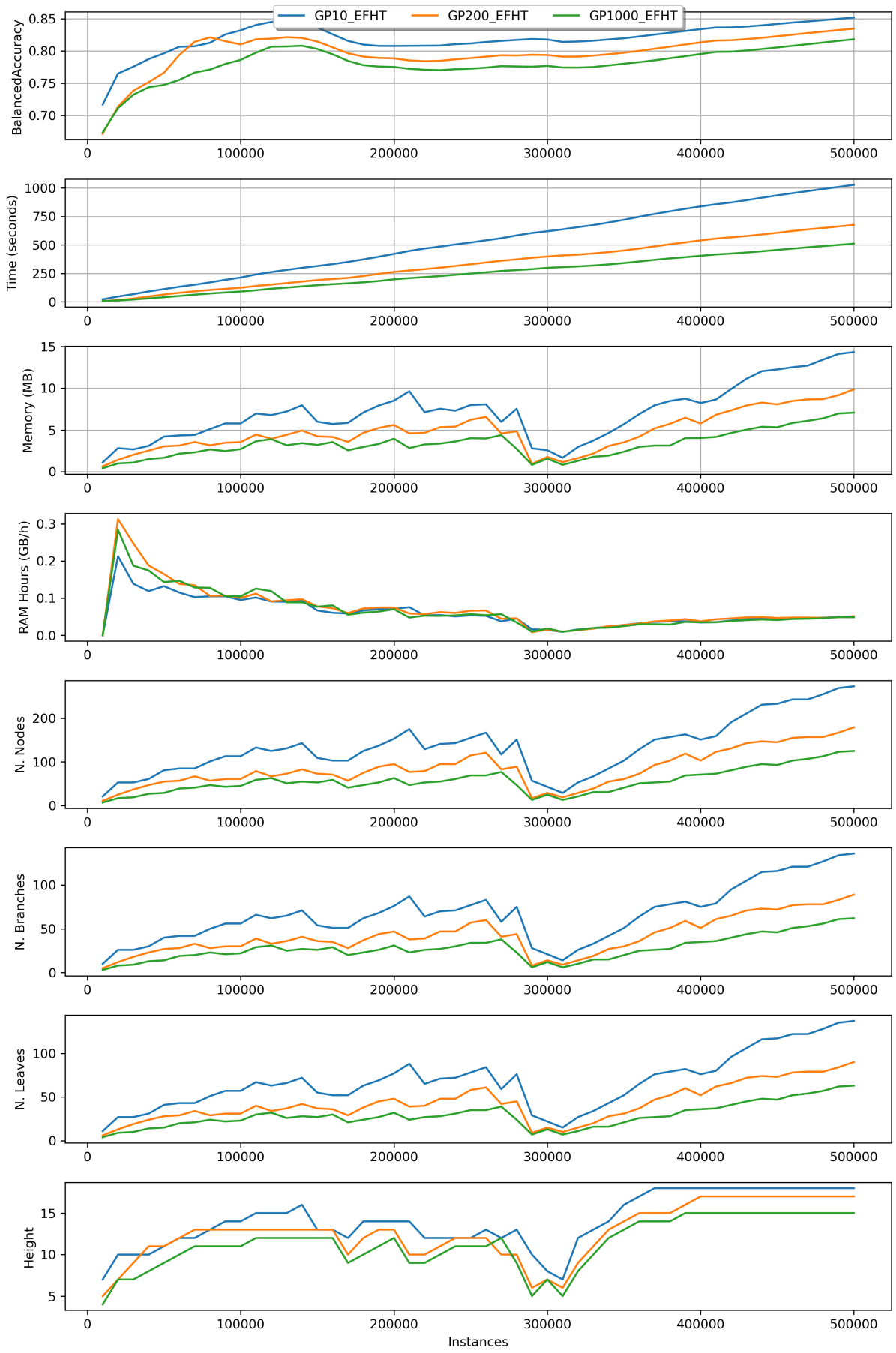


Figure 6.5: (River - Grace Period) Agrawal with 500K instances - EFDT

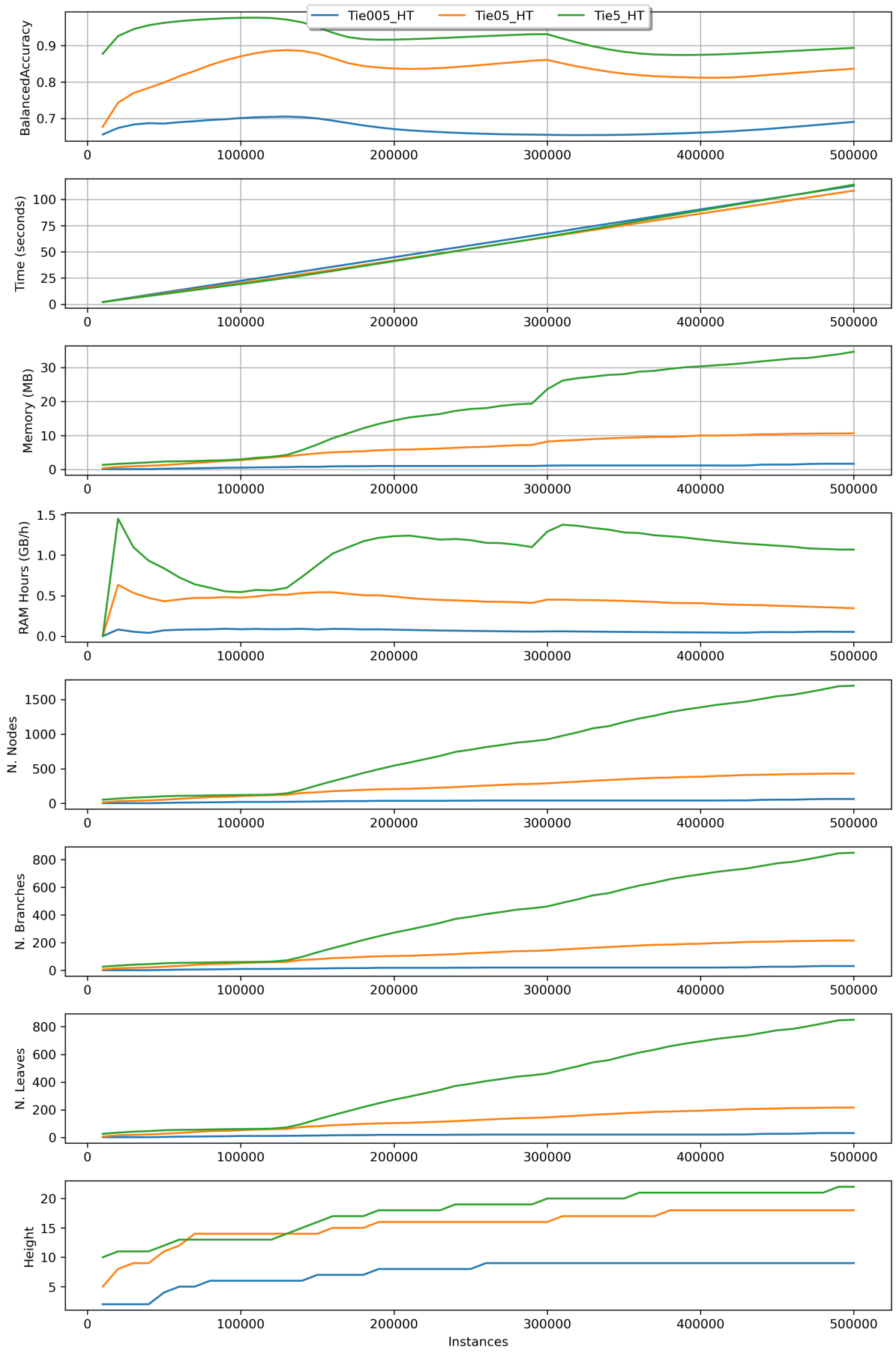


Figure 6.6: (River - Tie Treshold) Agrawak with 500K instances - HT

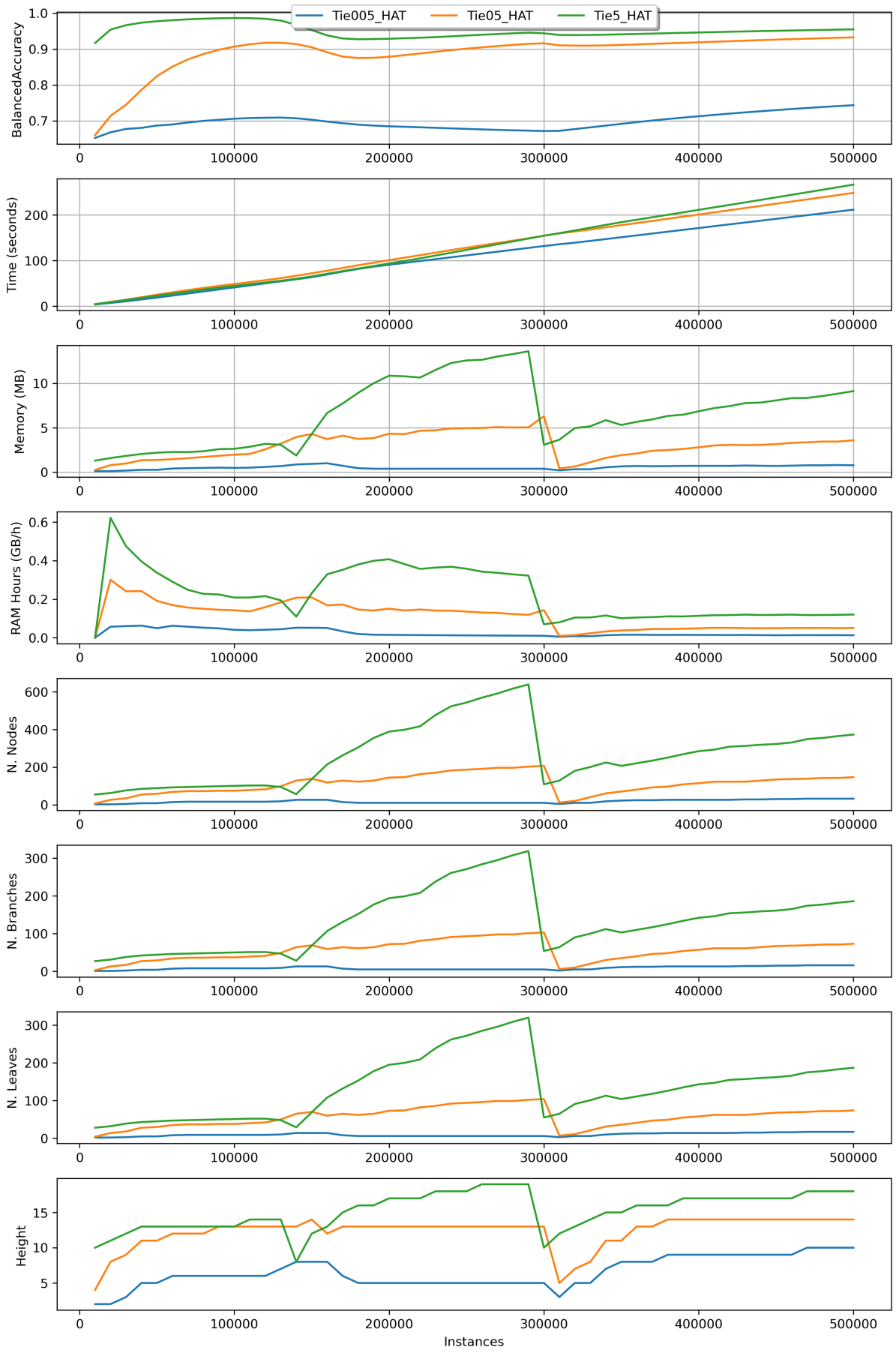


Figure 6.7: (River - Tie Threshold) Agrawak with 500K instances - HAT

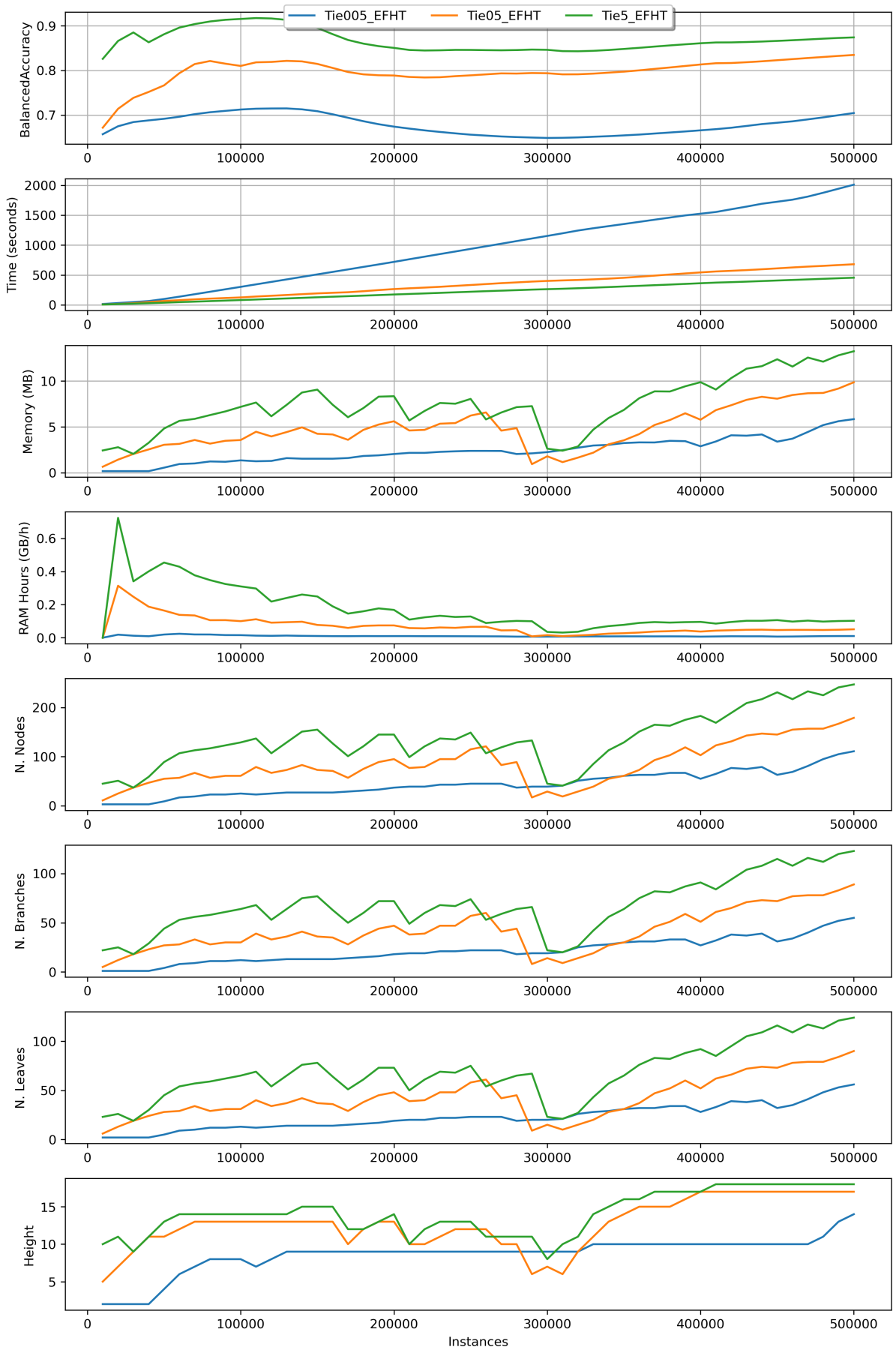


Figure 6.8: (River - Tie Treshold) Agrawak with 500K instances - EFDT

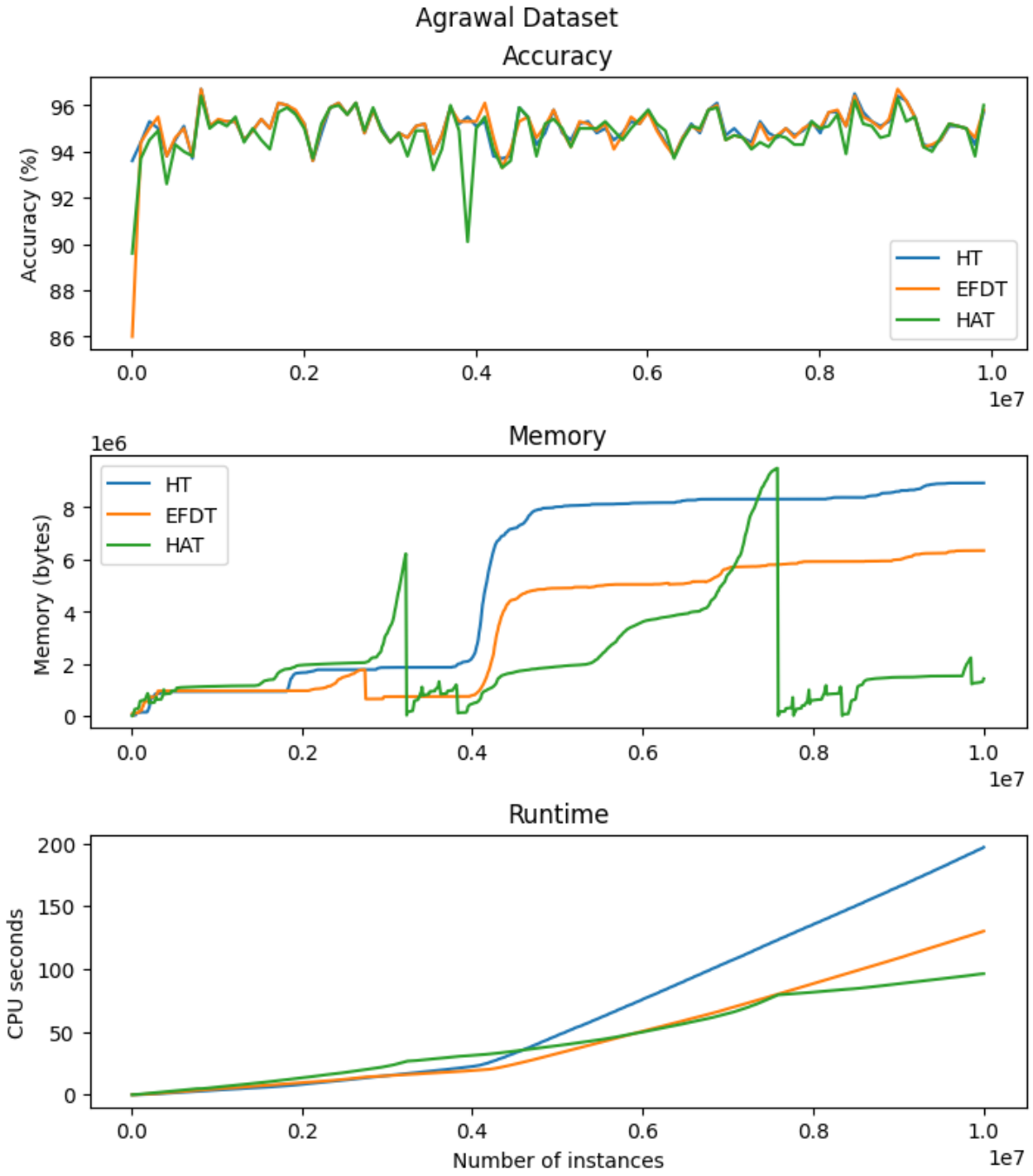


Figure 6.9: (MOA) Agrawal with 10M - HT - EFDT - HAT

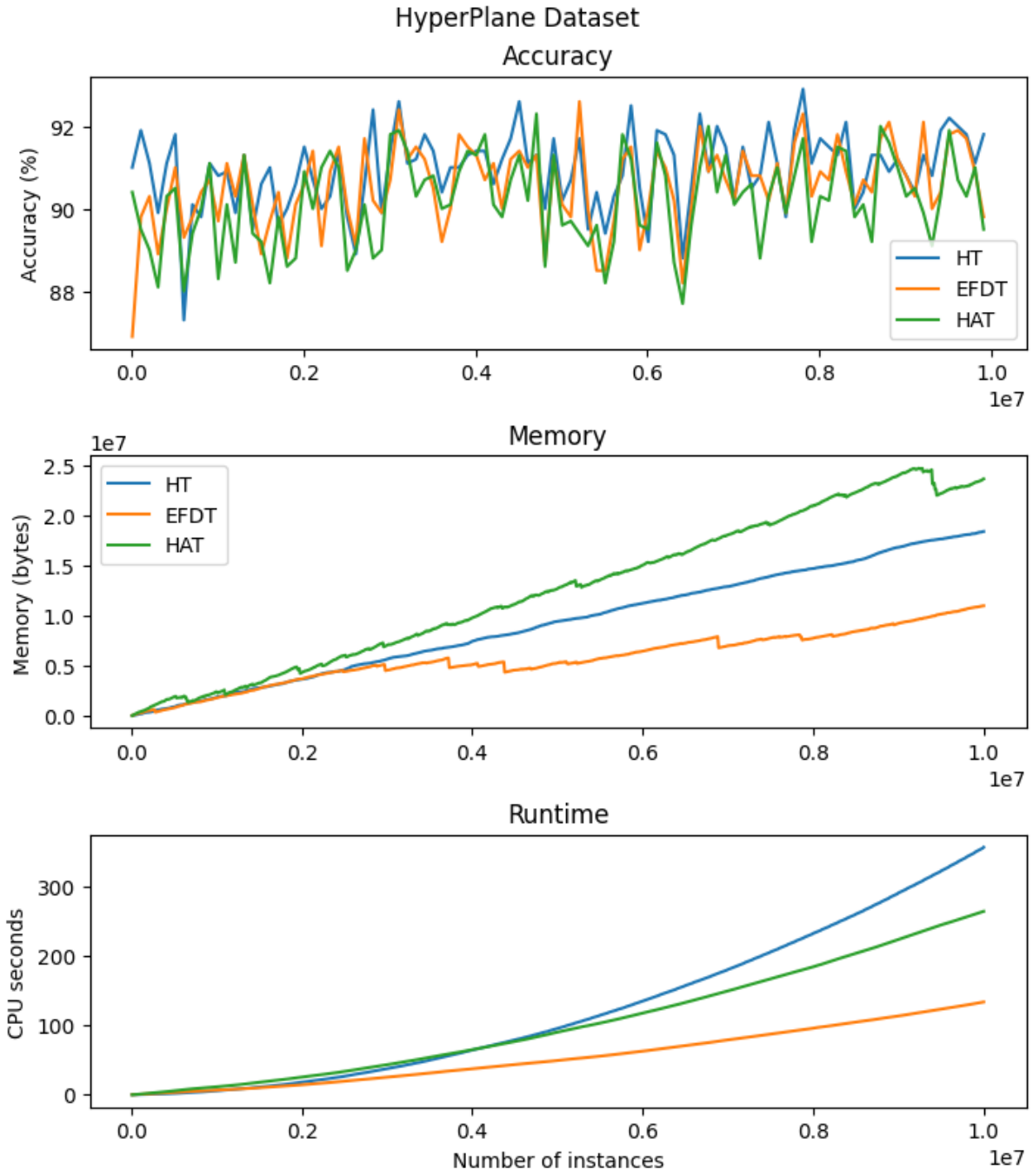


Figure 6.10: (MOA) Hyperplane with 10M - HT - EFDT - HAT

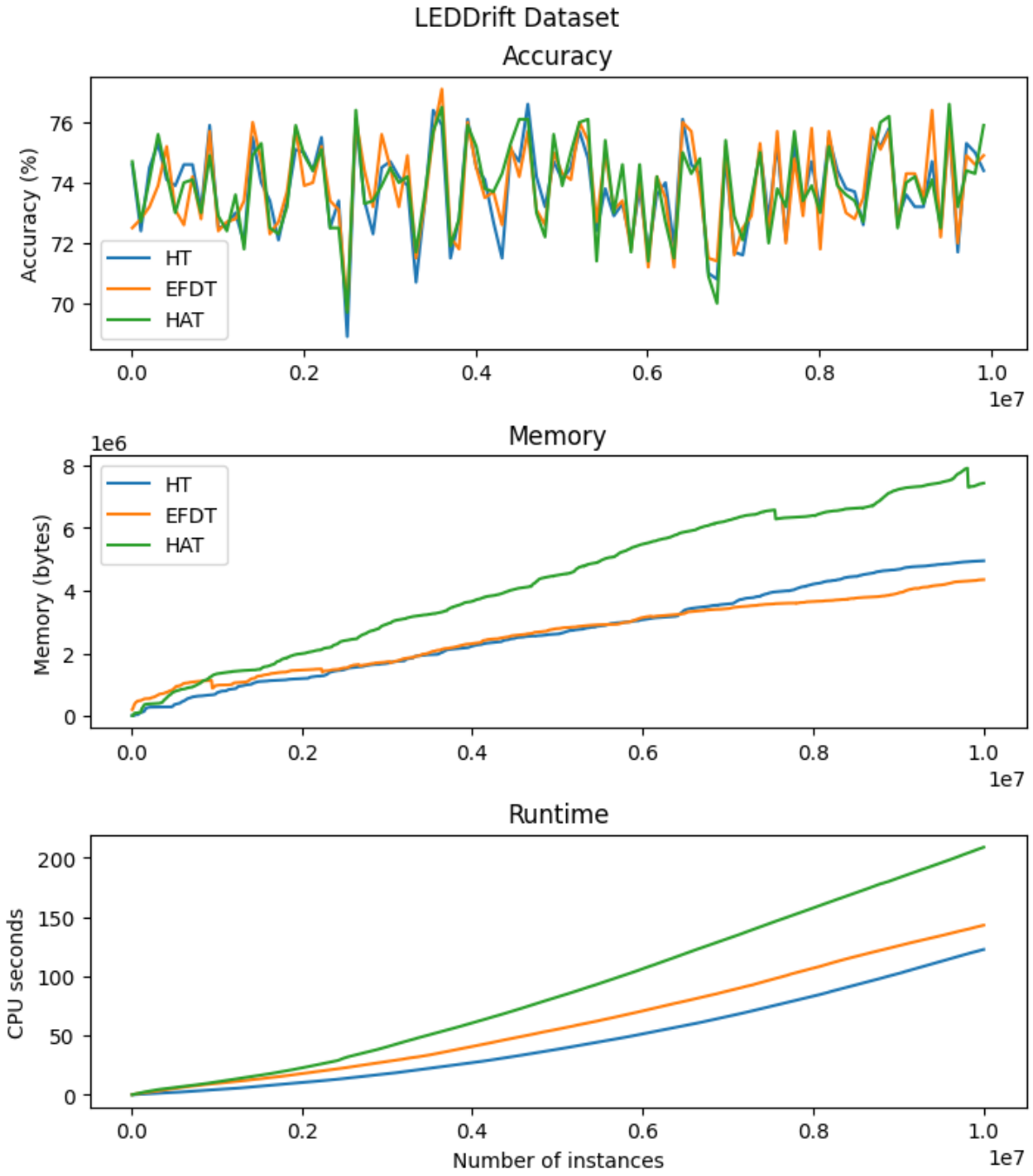


Figure 6.11: (MOA) LEDDrift with 10M - HT - EFDT - HAT

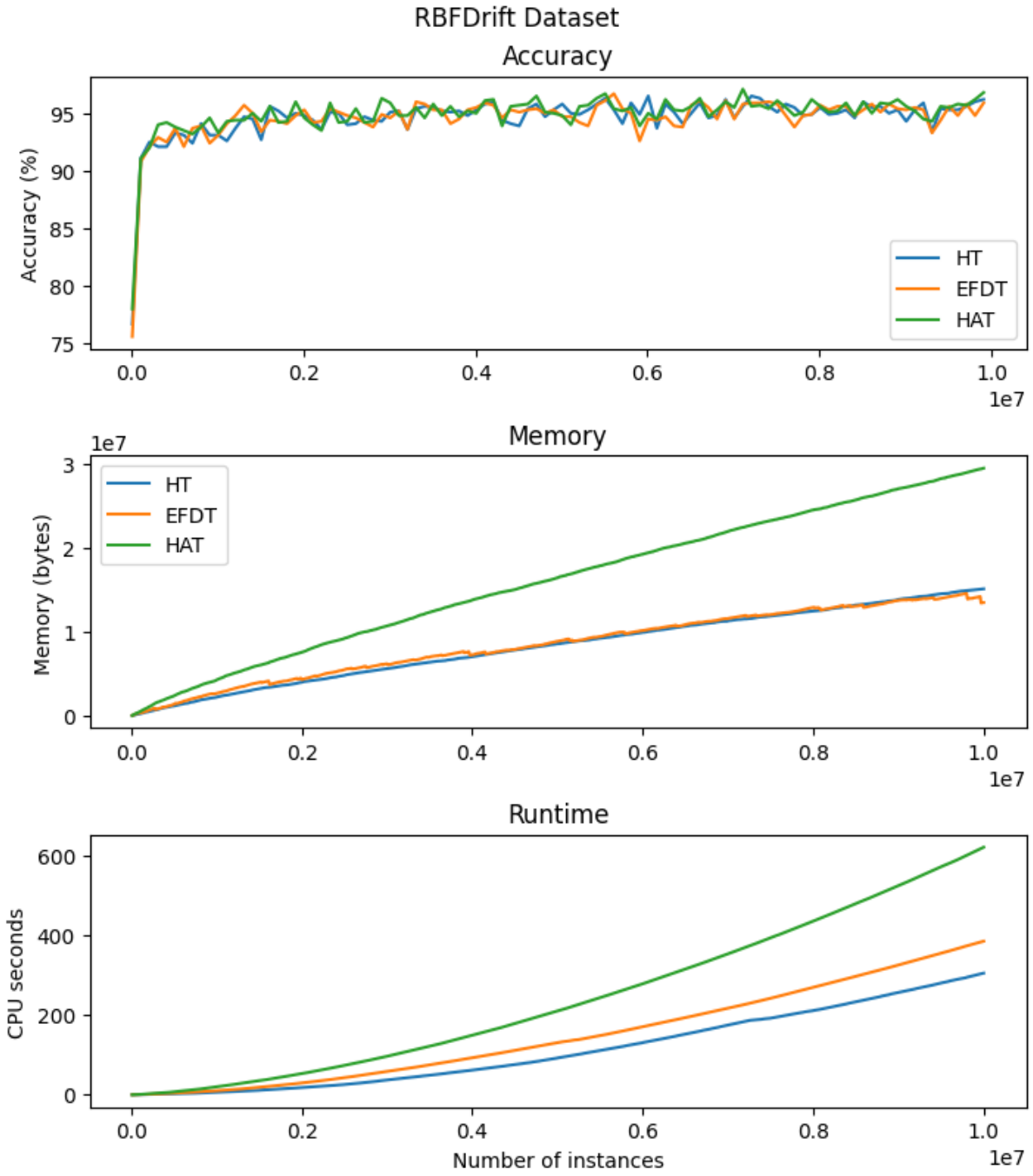


Figure 6.12: (MOA) RBFDrift with 10M - HT - EFDT - HAT

| Algorithm | Dataset | Evaluation Time (CPU s) | Model Cost (RAM-Hours) | Classifications Correct (%) | Kappa Statistic (%) |
|---|---|---|---|---|---|
| HT | Agrawal | 197 | 0.000379 | **95.9** | **90.76** |
| EFDT | Agrawal | 130 | 0.000162 | **95.9** | **90.76** |
| HAT | Agrawal | **96.3** | **0.0000761** | 95.8 | 90.55 |
| HT | HyperPlane | 356 | 0.00112 | 90.4 | 80.80 |
| EFDT | HyperPlane | **133** | **0.000225** | 90.3 | 80.59 |
| HAT | HyperPlane | 264 | 0.00105 | **90.6** | **81.18** |
| HT | LEDDrift | **123** | **0.000104** | 75.2 | 72.44 |
| EFDT | LEDDrift | 143 | 0.000106 | 75.2 | 72.44 |
| HAT | LEDDrift | 209 | 0.000264 | **75.3** | **72.55** |
| HT | RBFDrift | **305** | **0.000802** | **95.2** | **90.40** |
| EFDT | RBFDrift | 385 | 0.00101 | 94.7 | 89.40 |
| HAT | RBFDrift | 621 | 0.00309 | 95.1 | 90.20 |

Figure 6.13: Performance Metrics of HT, EFDT, and HAT Algorithms on Various Datasets (Part 1)

| Algorithm | Dataset | Model Serialized Size (bytes) | Tree Size (Nodes) | Tree Size (Leaves) | Active Learning Leaves | Tree Depth |
|---|---|---|---|---|---|---|
| HT | Agrawal | 8.93M | 2,395 | 2,046 | 2,046 | 11 |
| EFDT | Agrawal | 6.35M | **1,820** | **1,441** | **1,424** | 12 |
| HAT | Agrawal | **1.44M** | 5,717 | 4,452 | 4,452 | **8** |
| HT | HyperPlane | 18.37M | 6,579 | 3,290 | 3,290 | 18 |
| EFDT | HyperPlane | **10.97M** | **5,178** | **1,937** | **1,831** | **15** |
| HAT | HyperPlane | 23.61M | 8,191 | 3,397 | 3,397 | 16 |
| HT | LEDDrift | 4.96M | 649 | 325 | 325 | **11** |
| EFDT | LEDDrift | **4.35M** | **600** | **285** | **275** | 12 |
| HAT | LEDDrift | 7.43M | 802 | 361 | 361 | 12 |
| HT | RBFDrift | 15.12M | **5,497** | 2,749 | 2,749 | 36 |
| EFDT | RBFDrift | **13.49M** | 6,411 | **2,401** | **2,215** | **31** |
| HAT | RBFDrift | 29.48M | 6,837 | 3,419 | 3,419 | 33 |

Figure 6.14: Performance Metrics of HT, EFDT, and HAT Algorithms on Various Datasets (Part 2)

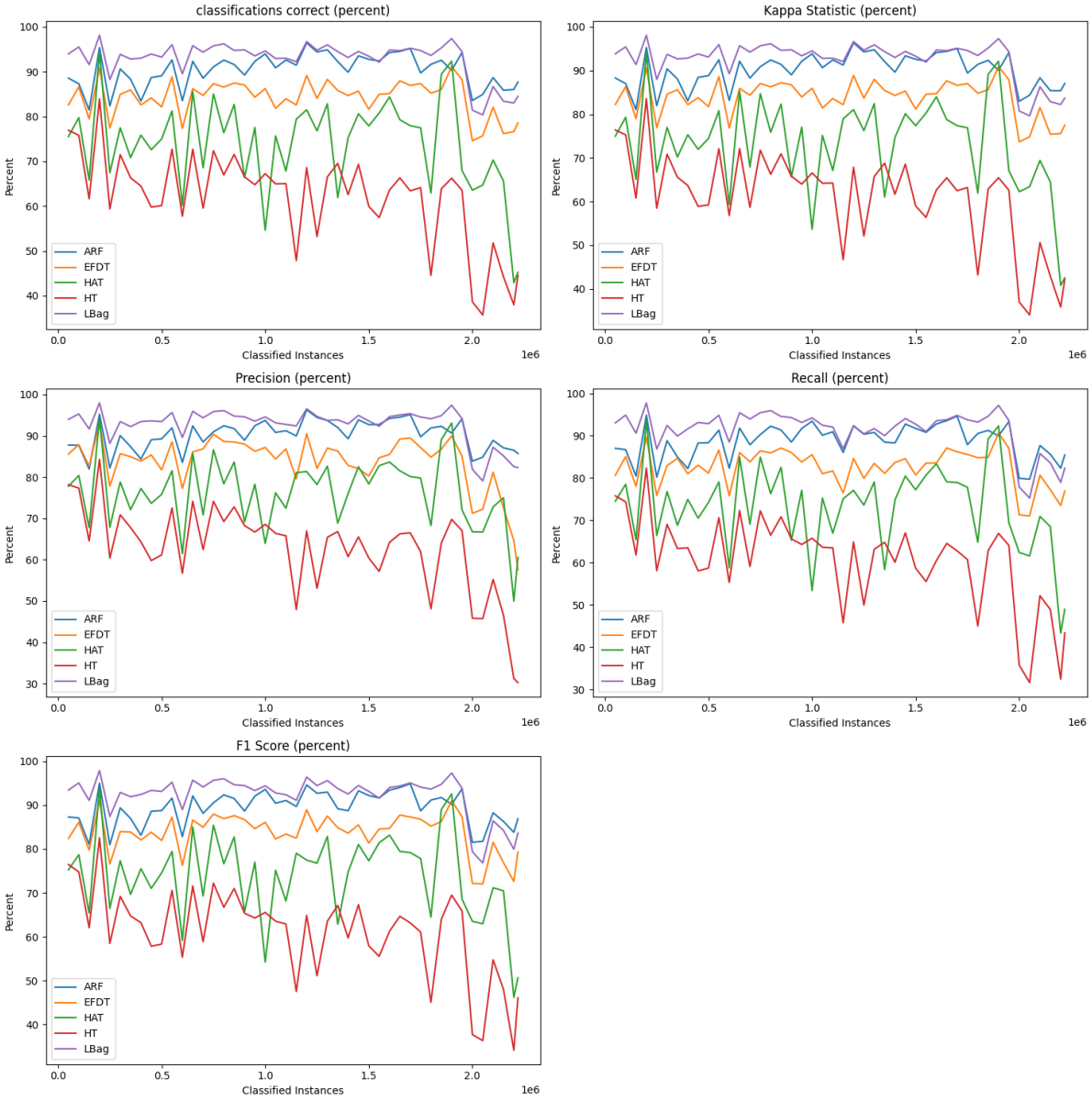


Figure 6.15: (CapyMOA - Real) Sensor with 2,219,803 instances - HT - HAT - EFDT - LBag - ARF

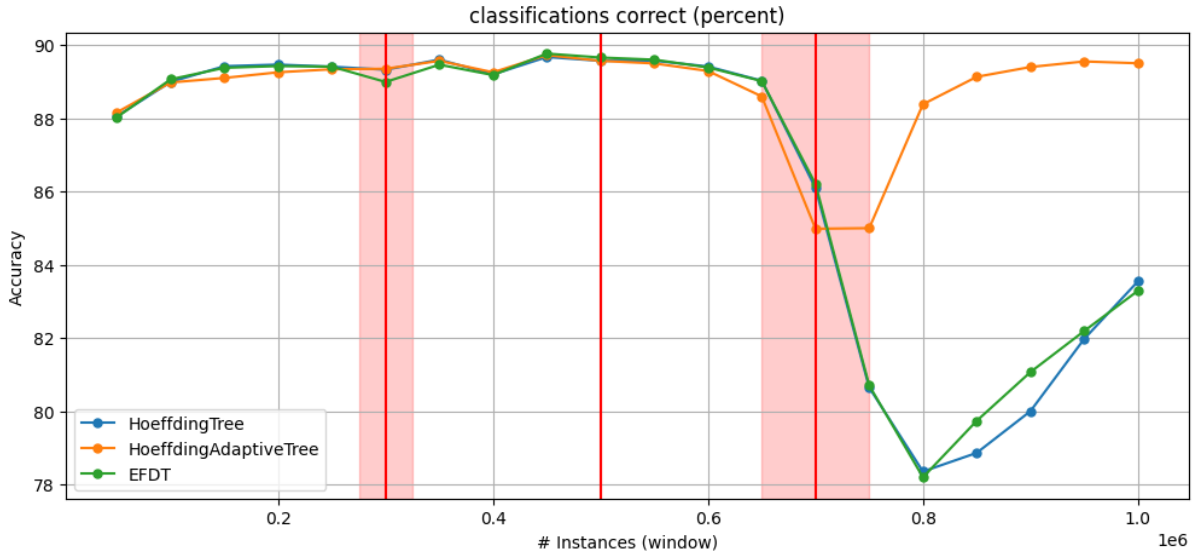


Figure 6.16: (CapyMOA - Synth) Sensor with 1M instances - HT - HAT - EFDT - LBag - ARF - SRP

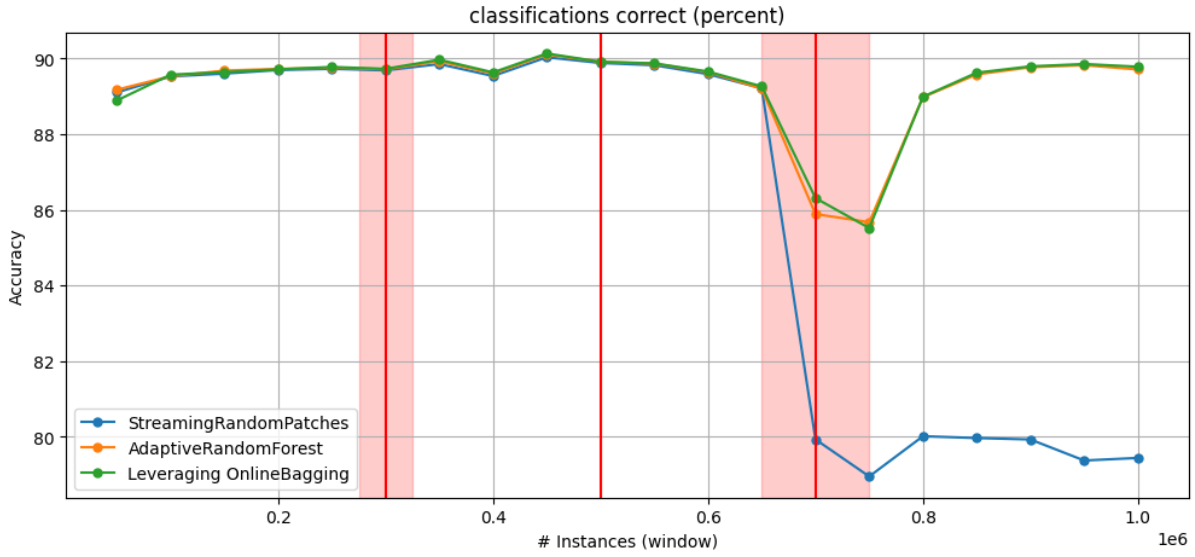


Figure 6.17: (CapyMOA) Esemble HT - (SEA 1M)

# Chapter 7

# Conclusions

The objectives of this thesis and its research questions were answered successfully. The main insights from this thesis can be described as follows:

- **Review of Data Processing Workflow**: We provided an in-depth review of the data processing workflow in the Internet of Things (IoT), focusing on data stream processing and its evolution, highlighting key issues such as latency. Additionally, we examined the transition to edge nodes, where data processing is performed closer to the data source to reduce latency and improve efficiency (Chapter 3).
- **Dynamic Between Batch and Stream Processing**: We explained the dynamics between batch and stream processing, detailing how these two methods can be used together to balance the need for real-time insights and comprehensive historical analysis (Chapter 3).
- **Introduction of Online Decision Trees (ODT)**: We introduced Online Decision Trees, which apply incremental learning to adapt to data streams. Key concepts such as dynamic structure, concept drift handling, and the importance of hyperparameters like Grace Period and Tie Threshold were highlighted (Chapter 4.)
- **Optimization Areas of ODT**: We described various optimization areas for Online Decision Trees, including concept drift detection and adaptation techniques, memory management techniques, improved splitting rules, and adaptive parameters. These techniques are crucial for maintaining the efficiency and accuracy of ODTs in memory-constrained environments (Chapter 5).
- **Experimental Evaluation**: Our experiments presented different frameworks dedicated to data stream mining. We evaluated the tunability and effects of Grace Period and Tie Threshold and compared various algorithms using synthetic and real-world datasets, including single decision trees and ensembles (Chapter 6).

This thesis offers a detailed overview of Online Decision Trees, particularly those based on the Hoeffding

Tree algorithm. This area's extensive and recent research indicates its ongoing evolution and potential for significant advancements, especially for resource-constrained edge devices.

One of the primary challenges encountered in this thesis was the implementation of experiments across different algorithms. Algorithms implemented in C++ or Cython, such as SVFDT, posed comparison challenges due to differing frameworks and implementation languages. Additionally, testing on larger datasets, particularly with ensemble methods, was constrained by the memory limitations of the experimental device.

Future work would be to develop a novel tree algorithm that integrates features from multiple existing trees. This was initially part of our research plan but was hindered by technical difficulties in modifying the SVFDT code implemented in C++ with Cython. The proposed algorithm would use SVFDT as a base, adjusting its expansion method to alternate between SVFDT I and II with criteria similar to what is used in GAHT. It would also incorporate adaptive Tie Thresholds and re-evaluation periods to reassess splits and handle concept drift effectively.

In conclusion, this research will continue under GECAD, to provide a deeper review of Online Decision Trees and develop a dedicated paper to propose the Enhanced SVFDT. The ongoing work aims to refine and enhance the current understanding and capabilities of Online Decision Trees, contributing to the broader field of data stream mining.

# Bibliography


[1] C. C. Aggarwal et al. “Social Network Stream Clustering”. In: *Managing and Mining Graph Data*. Springer, 2012, pp. 477–501.

[2] H. Becker et al. “Event Detection in Social Media”. In: *Proceedings of the ACM SIGIR Conference on Research and Development in Information Retrieval*. 2011.

[3] András A Benczúr, Levente Kocsis, and Róbert Pálovics. “Online machine learning in big data streams”. In: *arXiv preprint arXiv:1802.05872* (2018).

[4] Albert Bifet and Ricard Gavalda. “Adaptive learning from evolving data streams”. In: *Advances in Intelligent Data Analysis VIII: 8th International Symposium on Intelligent Data Analysis, IDA 2009, Lyon, France, August 31-September 2, 2009. Proceedings 8*. Springer. 2009, pp. 249–260.

[5] Albert Bifet and Ricard Gavalda. “Learning from time-changing data with adaptive windowing”. In: *Proceedings of the 2007 SIAM international conference on data mining*. SIAM. 2007, pp. 443–448.

[6] Albert Bifet, Geoff Holmes, and Bernhard Pfahringer. “Leveraging Bagging for Evolving Data Streams”. In: *Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2010, Barcelona, Spain, September 20-24, 2010, Proceedings, Part I*. Springer Berlin Heidelberg, 2010, pp. 135–150. DOI: 10.1007/978-3-642-15880-3_15.

[7] Albert Bifet et al. “Extremely fast decision tree mining for evolving data streams”. In: *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2017, pp. 1733–1742.

[8] Albert Bifet et al. “Fast perceptron decision tree learning from evolving data streams”. In: *Advances in Knowledge Discovery and Data Mining: 14th Pacific-Asia Conference, PAKDD 2010, Hyderabad, India, June 21-24, 2010. Proceedings. Part II 14*. Springer. 2010, pp. 299–310.

[9] Albert Bifet et al. “MOA: Massive Online Analysis”. In: *J. Mach. Learn. Res.* 11 (2010), pp. 1601–1604. DOI: 10.5555/1756006.1859903. URL: https://dl.acm.org/doi/10.5555/1756006.1859903.

[10] Vadim Borisov et al. “Deep neural networks and tabular data: A survey”. In: *IEEE transactions on neural networks and learning systems* (2022).

[11] Leo Breiman. *Classification and regression trees*. Routledge, 2017.

[12] Gonçalo Carvalho et al. "Edge computing: current trends, research challenges and future directions". In: *Computing* 103.5 (2021), pp. 993–1023.

[13] Kuan-Yu Chen et al. "Trompt: Towards a better deep neural network for tabular data". In: *arXiv preprint arXiv:2305.18446* (2023).

[14] S. Chien and N. Immorlica. "Temporal Correlation Analysis". In: *Proceedings of the ACM SIGIR Conference on Research and Development in Information Retrieval*. 2005.

[15] Victor Guilherme Turrisi da Costa, André Carlos Ponce de Leon Ferreira, Sylvio Barbon Junior, et al. "Strict very fast decision tree: a memory conservative algorithm for data stream mining". In: *Pattern Recognition Letters* 116 (2018), pp. 22–28.

[16] Vinícius G Costa and Carlos E Pedreira. "Recent advances in decision trees: An updated survey". In: *Artificial Intelligence Review* 56.5 (2023), pp. 4765–4800.

[17] 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*. 2000, pp. 71–80.

[18] Pietro Ducange, Francesco Marcelloni, Riccardo Pecori, et al. "Fuzzy Hoeffding decision tree for data stream classification." In: *Int. J. Comput. Intell. Syst.* 14.1 (2021), pp. 946–964.

[19] Lachit Dutta and Swapna Bharali. "Tinyml meets iot: A comprehensive survey". In: *Internet of Things* 16 (2021), p. 100461.

[20] Marta Fernandes, Juan Manuel Corchado, and Goreti Marreiros. "Machine learning techniques applied to mechanical fault diagnosis and fault prognosis in the context of real industrial manufacturing use-cases: a systematic literature review". In: *Applied Intelligence* 52.12 (2022), pp. 14246–14280.

[21] Joao Gama, Pedro Medas, and Ricardo Rocha. "Forest trees for on-line data". In: *Proceedings of the 2004 ACM symposium on Applied computing*. 2004, pp. 632–636.

[22] Joao Gama, Ricardo Rocha, and Pedro Medas. "Accurate decision trees for mining high-speed data streams". In: *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*. 2003, pp. 523–528.

[23] Eva Garcia-Martin, Niklas Lavesson, and Håkan Grahn. "Identification of energy hotspots: A case study of the very fast decision tree". In: *Green, Pervasive, and Cloud Computing: 12th International Conference, GPC 2017, Cetara, Italy, May 11-14, 2017, Proceedings 12*. Springer. 2017, pp. 267–281.

[24] Eva Garcia-Martin et al. "Green accelerated Hoeffding tree". In: *arXiv preprint arXiv:2205.03184* (2022).

[25] Heitor M. Gomes et al. “Adaptive Random Forests for Evolving Data Stream Classification”. In: *Machine Learning* 106.9-10 (2017), pp. 1469–1495. DOI: 10.1007/s10994-017-5642-8.

[26] Heitor M Gomes et al. “Adaptive random forests for evolving data stream classification”. In: *Machine Learning* 106 (2017), pp. 1469–1495.

[27] Heitor Murilo Gomes, Jesse Read, and Albert Bifet. “Streaming random patches for evolving data stream classification”. In: *2019 IEEE international conference on data mining (ICDM)*. IEEE. 2019, pp. 240–249.

[28] Léo Grinsztajn, Edouard Oyallon, and Gaël Varoquaux. “Why do tree-based models still outperform deep learning on typical tabular data?” In: *Advances in neural information processing systems* 35 (2022), pp. 507–520.

[29] Tyler L Hayes and Christopher Kanan. “Online continual learning for embedded devices”. In: *arXiv preprint arXiv:2203.10681* (2022).

[30] Wassily Hoeffding. “Probability inequalities for sums of bounded random variables”. In: *The collected works of Wassily Hoeffding* (1994), pp. 409–426.

[31] T Ryan Hoens, Robi Polikar, and Nitesh V Chawla. “Learning from streaming data with concept drift and imbalance: an overview”. In: *Progress in Artificial Intelligence* 1 (2012), pp. 89–101.

[32] Geoffrey Holmes, Kirkby Richard, and Bernhard Pfahringer. “Tie-breaking in Hoeffding trees”. In: ECML/PKDD. 2005.

[33] Geoff Hulten, Laurie Spencer, and Pedro Domingos. “Mining time-changing data streams”. In: *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*. 2001, pp. 97–106.

[34] Rajalakshmi Krishnamurthi et al. “An overview of IoT sensor data processing, fusion, and analysis techniques”. In: *Sensors* 20.21 (2020), p. 6076.

[35] Arvind Kumar, Parminder Kaur, and Pratibha Sharma. “A survey on Hoeffding tree stream data classification algorithms”. In: *CPUH-Res. J* 1.2 (2015), pp. 28–32.

[36] R. Kumar et al. “Community Detection in Social Networks”. In: *Communications of the ACM* 53.6 (2010), pp. 102–109.

[37] Mai Lefa, ABD Hatem, and Rashed Salem. “Enhancement of Very Fast Decision Tree for Data Stream Mining”. In: *Studies in Informatics and Control* 31.2 (2022), pp. 49–60.

[38] Chao Li et al. “Edge-oriented computing paradigms: A survey on architecture design and system management”. In: *ACM Computing Surveys (CSUR)* 51.2 (2018), pp. 1–34.

[39] Viktor Losing, Heiko Wersing, and Barbara Hammer. “Enhancing very fast decision trees with local split-time predictions”. In: *2018 IEEE international conference on data mining (ICDM)*. IEEE. 2018, pp. 287–296.

[40] Jie Lu et al. “Learning under concept drift: A review”. In: *IEEE transactions on knowledge and data engineering* 31.12 (2018), pp. 2346–2363.

[41] Chaitanya Manapragada, Mahsa Salehi, and Geoffrey I Webb. “Extremely fast hoeffding adaptive tree”. In: *2022 IEEE International Conference on Data Mining (ICDM)*. IEEE. 2022, pp. 319–328.

[42] Chaitanya Manapragada, Geoffrey I Webb, and Mahsa Salehi. “Extremely fast decision tree”. In: *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2018, pp. 1953–1962.

[43] Jacob Montiel et al. “River: machine learning for streaming data in Python”. In: *Journal of Machine Learning Research* 22.110 (2021), pp. 1–8. URL: http://jmlr.org/papers/v22/20-1380.html.

[44] A. Moore et al. “Network Monitoring and Analysis”. In: *Proceedings of the ACM SIGCOMM Internet Measurement Conference*. 2003.

[45] James N Morgan and John A Sonquist. “Problems in the analysis of survey data, and a proposal”. In: *Journal of the American statistical association* 58.302 (1963), pp. 415–434.

[46] Asmah Muallem et al. “Hoeffding tree algorithms for anomaly detection in streaming datasets: A survey”. In: *Journal of Information Security* 8.4 (2017).

[47] H. L. Nguyen et al. “Data Stream Applications”. In: (2023).

[48] Bernhard Pfahringer, Geoffrey Holmes, and Richard Kirkby. “New options for hoeffding trees”. In: *AI 2007: Advances in Artificial Intelligence: 20th Australian Joint Conference on Artificial Intelligence, Gold Coast, Australia, December 2-6, 2007. Proceedings 20*. Springer. 2007, pp. 90–99.

[49] Tobias Pfandzelter and David Bermbach. “IoT data processing in the fog: Functions, streams, or batch processing?” In: *2019 IEEE International conference on fog computing (ICFC)*. IEEE. 2019, pp. 201–206.

[50] Leszek Rutkowski et al. “Decision trees for mining data streams based on the McDiarmid’s bound”. In: *IEEE Transactions on Knowledge and Data Engineering* 25.6 (2012), pp. 1272–1279.

[51] J. Sankaranarayanan et al. “Social Network Stream Classification”. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 2009.

[52] Nikolaos Schizas et al. “TinyML for ultra-low power AI and large scale IoT deployments: A systematic review”. In: *Future Internet* 14.12 (2022), p. 363.

[53] FM Javed Mehedi Shamrat et al. “A comprehensive study on pre-pruning and post-pruning methods of decision tree classification algorithm”. In: *2021 5th International conference on trends in electronics and informatics (ICOEI)*. IEEE. 2021, pp. 1339–1345.

[54] Weisong Shi et al. “Edge Computing: Vision and Challenges”. In: *IEEE Internet of Things Journal* 3.5 (2016), pp. 637–646. DOI: 10.1109/JIOT.2016.2579198.

[55] Jiang Sun et al. “Speeding up very fast decision tree with low computational cost”. In: *Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence*. 2021, pp. 1272–7278.

[56] J. Warren and N. Marz. *Big Data: Principles and best practices of scalable realtime data systems*. Manning, 2015. ISBN: 9781638351108.

[57] Fatos Xhafa, Burak Kilic, and Paul Krause. “Evaluation of IoT stream processing at edge computing layer for semantic data enrichment”. In: *Future Generation Computer Systems* 105 (2020), pp. 730–736. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2019.12.031. URL: https://www.sciencedirect.com/science/article/pii/S0167739X19321296.

[58] Hang Yang and Simon Fong. “Moderated VFDT in stream mining using adaptive tie threshold and incremental pruning”. In: *Data Warehousing and Knowledge Discovery: 13th International Conference, DaWaK 2011, Toulouse, France, August 29-September 2, 2011. Proceedings 13*. Springer. 2011, pp. 471–483.

[59] Hang Yang and Simon Fong. “Moderated VFDT in Stream Mining Using Adaptive Tie Threshold and Incremental Pruning”. In: *Data Warehousing and Knowledge Discovery*. Ed. by Alfredo Cuzzocrea and Umeshwar Dayal. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 471–483. ISBN: 978-3-642-23544-3.

[60] Hang Yang and Simon Fong. “Optimized very fast decision tree with balanced classification accuracy and compact tree size”. In: *The 3rd international conference on data mining and intelligent information technology applications*. IEEE. 2011, pp. 57–64.

[61] H. Zeng et al. “Clustering Web Search Results”. In: *Proceedings of the ACM SIGIR Conference on Research and Development in Information Retrieval*. 2004.

[62] Wenbin Zhang and Albert Bifet. “Feat: A fairness-enhancing and concept-adapting decision tree classifier”. In: *Discovery Science: 23rd International Conference, DS 2020, Thessaloniki, Greece, October 19–21, 2020, Proceedings 23*. Springer. 2020, pp. 175–189.

[63] Theo Zschörnig, Robert Wehlitz, and Bogdan Franczyk. “A personal analytics platform for the internet of things-implementing kappa architecture with microservice-based stream processing”. In: *International Conference on Enterprise Information Systems*. Vol. 2. SCITEPRESS. 2017, pp. 733–738.