UNIVERSIDADE DE SÃO PAULO
Instituto de Ciências Matemáticas e de Computação

Efficient online tree, rule-based and distance-based algorithms

Saulo Martiello Mastelini

Tese de Doutorado do Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional (PPG-CCMC)

ICMC USP SÃO CARLOS

Saulo Martiello Mastelini

Efficient online tree, rule-based and distance-based algorithms

Thesis submitted to the Instituto de Ciências Matemáticas e de Computação – ICMC-USP – in accordance with the requirements of the Computer and Mathematical Sciences Graduate Program, for the degree of Doctor in Science. *EXAMINATION BOARD PRESENTATION COPY*

Concentration Area: Computer Science and Computational Mathematics

Advisor: Prof. Dr. André Carlos Ponce de Leon Ferreira de Carvalho

USP – São Carlos
February 2023

Ficha catalográfica elaborada pela Biblioteca Prof. Achille Bassi
e Seção Técnica de Informática, ICMC/USP,
com os dados inseridos pelo(a) autor(a)

M378e

Martiello Mastelini, Saulo
Efficient online tree, rule-based and distance-based algorithms / Saulo Martiello Mastelini; orientador André Carlos Ponce de Leon Ferreira de Carvalho. -- São Carlos, 2023.
178 p.

Tese (Doutorado - Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional) -- Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, 2023.

1. Decision trees. 2. Decision rules. 3. Ensembles. 4. Online machine learning. 5. Data streams. I. Ponce de Leon Ferreira de Carvalho, André Carlos, orient. II. Título.

Bibliotecários responsáveis pela estrutura de catalogação da publicação de acordo com a AACR2:
Gláucia Maria Saia Cristianini - CRB - 8/4938
Juliana de Souza Moraes - CRB - 8/6176

Saulo Martiello Mastelini

Algoritmos incrementais e eficientes para árvores e regras de decisão e algoritmos baseados em proximidade

Tese apresentada ao Instituto de Ciências Matemáticas e de Computação – ICMC-USP, como parte dos requisitos para obtenção do título de Doutor em Ciências – Ciências de Computação e Matemática Computacional. *EXEMPLAR DE DEFESA*

Área de Concentração: Ciências de Computação e Matemática Computacional

Orientador: Prof. Dr. André Carlos Ponce de Leon Ferreira de Carvalho

USP – São Carlos
Fevereiro de 2023

To my parents, Edson and Vera. Every victory I have achieved so far was because you were there to help and root for me.

ACKNOWLEDGEMENTS

First and foremost, I would like to thank God for always showing me the way in some form and strengthening me when I thought I could not keep going anymore. Your kindness and compassion endure forever.

It is not easy to grasp the number of interactions and persons that, in any way, shape, or form, have contributed to the fulfillment of this thesis. After all, my Ph.D. dream began a long way before my first steps at ICMC-USP, back in August 2018. With this dream also came multiple supporters and people that aided me in arriving at São Carlos. To all of them, even though I might not even remember their role in this journey anymore, I am very grateful.

In special, I would like to thank my parents, Antonio Edson Mastelini and Vera Lucia Martiello Mastelini, for always being there and going above and beyond to fulfill my dreams. I know you two had to sacrifice many of your dreams and expectations so that I could pursue mine. I will always be thankful for that, no matter how much I try to compensate in any pointless way for all you have done for me. I dream of one day performing the same role and raising children with at least one-tenth of your dedication to my sister and me. If I can do that, I know that I will have been a good father.

Talking about my sister, I could not thank you either, Ana Paula Martiello Mastelini, for all the support and wise advice you have given me all these years. Although I am the older brother, you perform the role of the older sister and always show me how dumb some of my ideas and perceptions are. I hope one day to be able to help you as you did to me since always.

I would also like to thank my grandmother (*in memoriam*), Geni Luzia Vacario Martiello. With all your simplicity, you always rooted for me and made everything you could to help me. It saddens me you were not able to wait to see the end of this journey. During my stay in Londrina, you were a second mother to me. Thank you for always believing that through knowledge and science, I could go further. I hope you watch over us from up there, we miss you.

To my fiancee, Gabriela Drews Wayhs, you were not here since the start of this journey, but since you arrived, everything has changed. Life is better with you and now I dream again. The same dreams of my early youth. Your kindness and wisdom inspire me to be someone better. “We’ve only just begun”, and the future is bright for us! I pray the family we dream of starting will be rooted in both of our families. If we are able to do that, I am sure everything will be alright. I love you, pretty one.

I also want to thank all my friends. The ones I talk with frequently these days and the

ones I have made through the way and I might have lost contact temporarily. My friends from Cândido de Abreu, Londrina, São Carlos, Europe, and well, all around the world really, you were essential to my journey in different ways. The joy of friendship moves me. Citing names would not be fair since my “bugged” mind could forget to mention someone, and I would not easily forgive myself for that. My consolation is that I was already able to personally thank most of you for your role in my thesis and in my life. We will see each other someday, somewhere, soon, I hope.

Next, I would like to thank my advisor, André Carlos Ponce de Leon Ferreira de Carvalho, and the advisor of my internship abroad, João Gama. What I have learned from you goes way beyond the technical part. I will always remember what you have taught me and will try to become a better researcher and human being inspired by both of you. You were among the best, most humble, and kindest scientists I have met.

Lastly, I would like to acknowledge the financial support given by Fapesp through the grants #2018/07319-6 and #2021/10488-7.

"Remember that almost everything looks better after a good night's sleep."
(Life's Little Instruction Book)

RESUMO

MARTIELLO MASTELINI, S. **Algoritmos incrementais e eficientes para árvores e regras de decisão e algoritmos baseados em proximidade**. 2023. 175 p. Tese (Doutorado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2023.

O rápido desenvolvimento de tecnologias digitais acarretou a produção constante de grandes volumes de dados, que se apresentam em diferentes formas e vêm de diferentes fontes. No início dos estudos de aprendizado de máquina (AM) a escassez de dados era um problema relevante em muitos domínios de aplicação, atualmente, no entanto, pode-se ter informação em demasia para tratar com algoritmos tradicionais de AM. Além disso, mudanças ao longo do tempo na distribuição probabilística que governa o processo de geração dos dados podem fazer com que as soluções tradicionais de AM se tornem inúteis em aplicações do mundo real. AM *online* (AMO) é uma área de estudos que busca criar soluções capazes de processar os dados incrementalmente, utilizando recursos computacionais limitados e lidando com distribuições de dados que mudam no decorrer do tempo. Apesar de a literatura em AMO apresentar soluções eficientes que foram aplicadas em domínios de aplicação diversos, existe uma tendência crescente de se criar algoritmos que focam apenas no desempenho preditivo, deixando o custo computacional em segundo plano. Essa observação é ainda mais predominante quando se considera tarefas de regressão que utilizam árvores e regras de decisão, bem como *ensembles* desses modelos, que estão dentre as soluções mais populares em AMO. Diminuir o custo computacional de soluções de AMO, de um ponto de vista do domínio de aplicação, pode ser mais relevante do que obter um leve aumento no desempenho preditivo. Assim, nessa tese, busca-se criar algoritmos de AMO cujo maior foco é a redução do tempo de processamento e do uso de memória em soluções de regressão baseadas em árvores e regras de decisão, além de *ensembles* formados por esses tipos de modelos. Um subproduto desejado é melhorar, ou pelo menos não impactar negativamente, o desempenho preditivo dos modelos. Na tese também é explorado um algoritmo eficiente para realizar buscas por vizinhos mais próximos de forma incremental. A tese é organizada como uma coleção de artigos, que compreende as publicações mais relevantes focadas nos temas apresentados. São abordadas estratégias para criar *ensembles* de regressão com baixo erro preditivo, propostos algoritmos eficientes de regressão incremental baseados em árvores de decisão, bem como um algoritmo para criação de *ensembles* baseados em árvores de decisão para regressão com baixo custo computacional e baixo erro preditivo. Por fim, é apresentado um algoritmo rápido e versátil para realizar buscas por vizinho mais próximo em janelas deslizantes de dados.

Palavras-chave: Aprendizado de máquina incremental, aprendizado supervisionado, desempenho computacional, regressão, busca por vizinhos mais próximos. .

ABSTRACT

MARTIELLO MASTELINI, S. **Efficient online tree, rule-based and distance-based algorithms**. 2023. 175 p. Tese (Doutorado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2023.

The fast development of digital technologies has given rise to the constant production of data in different forms and from different sources. While at the beginning of machine learning (ML) studies, data scarcity was a relevant problem for many application domains, nowadays, we may have too much information to handle with traditional ML algorithms. Besides, changes in the underlying data distributions that govern the data generation might render traditional ML solutions useless in real-world applications. Online ML (OML) aims to create solutions able to process data incrementally, with limited computation resource usage, and to deal with time-changing data distributions. Despite successfully creating efficient solutions applied in diverse domains, we have seen a recent growing trend in creating OML algorithms that only focus on predictive performance and overlook computational costs. This observation is even more prevalent when considering regression tasks, using decision trees, decision rules, and ensembles thereof, which are among the most popular OML solutions. Decreasing the computational costs of OML solutions could be more relevant than a slight increase in predictive performance from a real-world application standpoint. Hence, in this thesis, we focus on creating improved and efficient OML algorithms whose primary focus is decreasing the time and memory costs of tree and decision rule-based regressors and ensemble-based regressors. The desired bi-product is improving or, at least, leaving the predictive performance unchanged. We also explore an efficient algorithm to perform incremental nearest-neighbor searches. This thesis is organized as an article collection, comprehending our most relevant publications focused on the presented theme. We tackle strategies to create low-error ensemble-based regressors, efficient strategies to build incremental decision tree regressors, propose a fast and accurate decision tree-based ensemble regressor, and explore an efficient and versatile algorithm to perform nearest neighbor search in sliding windows.

Keywords: Online machine learning, Supervised learning, Computational performance, Regression, Nearest Neighbor Search .

LIST OF FIGURES

LIST OF ALGORITHMS

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

ADWIN	ADaptive WINdow
AMRules	Adaptive Model Rules
AO	Attribute Observers
ARF	Adaptive Random Forest
BAG	Bagging
CART	Classification and Regression Tree
DT	Decision Tree
E-BST	Extended Binary Search Tree
FIMT-DD	Fast and Incremental Model Trees
HB	Hoeffding Bound
HT	Hoeffding Trees
HTR	Hoeffding Tree Regressors
LS	Linear Scan
MAE	Mean Absolute Error
ML	Machine Learning
MSE	Mean Squared Error
NNS	Nearest neighbor search
OML	Online Machine Learning
OXT	Online Extra Trees
PAR	Passive-Aggressive Regression
PCA	Principal Component Analysis
QO	Quantization Observer
R^2	Coefficient of determination
RF	Random Forest
RMSE	Root Mean Square Error
RP	Random Patches
RS	Random Subspaces
SR	Search Recall
SRP	Streaming Random Patches
SWINN	Sliding Window-based Incremental Nearest Neighbors
VR	Variance Reduction
XT	Extra Trees

CONTENTS

CHAPTER

1

INTRODUCTION

The last decades have brought a high increase in data production. This expansion in data availability also increased the demand for automation with technological and innovative solutions able to make intelligent decisions (GAMA, 2010; RUSSELL; NORVIG, 2016). Intelligence, in this context, implies being able to mimic in some form the decision process a human, possibly a field specialist, would follow to find a solution for a given problem. Indeed, replacing human activities in tedious, fatigue-prone, or dangerous situations can increase productivity, reduce mistakes and enable specialists to focus on more critical activities (WANG; SIAU, 2019). Naturally, Machine Learning (ML) has arisen as a popular solution for such demands and has been applied to a broad selection of areas, ranging from material sciences (MASTELINI *et al.*, 2022) and biology (LIBBRECHT; NOBLE, 2015) to human resource management (GARG *et al.*, 2021), among others (RUSSELL; NORVIG, 2016; SARKER, 2021).

Traditional or batch-based ML algorithms expect a fixed amount of training data available to induce the learning models (GAMA, 2010). Once trained, such models can be applied to provide predictions. In many practical situations, data is continuously produced, thus potentially arriving indefinitely. Batch ML algorithms are not well suited to process such massive amounts of data due to computational restrictions (BIFET *et al.*, 2018). A trivial alternative is to select data samples for model training by assuming they represent the data distribution. Although a reasonable assumption broadly adopted in practical applications, this strategy does not cope with time-changing distributions. Concept drift (GAMA *et al.*, 2014; LU *et al.*, 2018), i.e., changes in the underlying data distribution, can influence the performance, robustness, and validity of trained models. Hence, ML algorithms able to learn and adapt themselves incrementally are desired in a scenario where data is continually produced with high throughput. Creating such models is the primary goal of Online Machine Learning (OML), which is one of the main tools used for data stream mining.

According to Bifet and Gavaldà (2009), the desiderata for OML solutions are:

- to process an example at a time and inspect it only once (at most);
- to be ready to predict at any point;
- to be aware that data may be evolving over time;
- to expect an infinite stream but process it under finite resources (time and memory).

By following these guidelines, there was a constant search for creating new algorithms able to process data streams. When analyzing the development in this area reported by recent literature, many researchers successfully created OML models with high predictive performance, e.g., Gomes *et al.* (2021). Many of these works created ensembles of OML models and relied on Decision Trees (DTs) as building blocks. In fact, ensemble learning is a trend in batch ML and OML (KRAWCZYK *et al.*, 2017; SAGI; ROKACH, 2018; GOMES *et al.*, 2021). Reducing the computational costs of incremental DTs and similar decision rules (DRs) induction algorithms have been addressed for classification tasks (PFAHRINGER; HOLMES; KIRKBY, 2008). These improvements directly benefit all the DT-based ensemble classifiers. Historically, multiple popular OML solutions were proposed for classification tasks and later adapted for regression, e.g., Domingos and Hulten (2000) and Ikonomovska, Gama and Džeroski (2011b). This observation is also accurate when considering ensembles of OML models, though regression has historically received less attention from the research community compared to classification.

Despite the efforts presented, for instance, in Pfahringer, Holmes and Kirkby (2008), computational performance is often overlooked, though it is a crucial element for OML algorithms. This observation is even more prevalent when considering incremental regression. Contrary to the need for more computationally efficient OML solutions, we have seen a frequent number of proposals of algorithms that seek to increase, ever so slightly, predictive performance with a not uncommon increase in computational costs (KORYCKI; KRAWCZYK, 2020; CANO; KRAWCZYK, 2022).

In this thesis, this is the main motivation to decide to take the opposite path of the current trends and focus on creating solutions better suited to real-world applications where computation resources are usually limited. Therefore, in this thesis, we mainly focused on improving the running time and memory usage of tree and decision rule-based regressors. Increasing the predictive performance was also a desired bi-product. We give a general view of our main contributions in section 1.3 and present a timeline of our proposal and how they complement each other in section 1.4. Next, we describe the basic aspects of algorithms that induce DTs, DRs, and ensembles of these models, which are necessary to better comprehend the contents presented in this thesis.

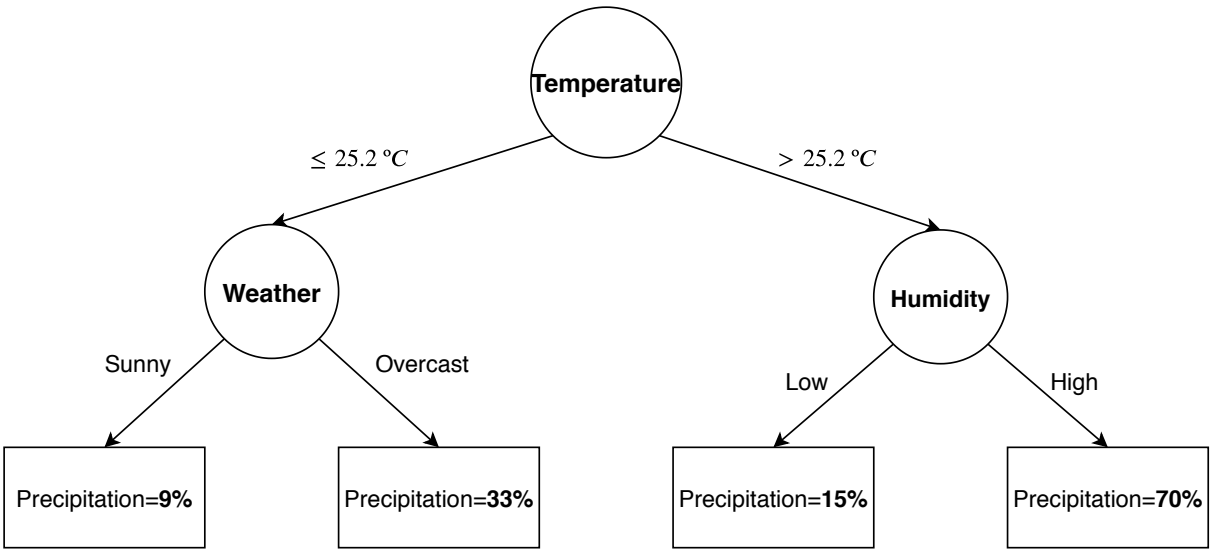
1.1 Decision trees, decision rules, and ensembles thereof

DTs induction algorithms are a popular family of ML algorithms applied to multiple learning tasks (LOH, 2011; BISHOP, 2006; GAMA, 2010; RUSSELL; NORVIG, 2016; BREIMAN *et al.*, 2017). They operate by recursively partitioning the input space, typically by performing a sequence of tests, one predictive attribute at a time (HO, 1995; BREIMAN, 2001; BREIMAN *et al.*, 2017). However, other alternatives can be explored, such as applying option nodes (KOHAVI; KUNZ, 1997; PFAHRINGER; HOLMES; KIRKBY, 2007). DTs fit a simple decision model within each of their inner partitions.

For such, they follow the "divide-and-conquer" principle (BREIMAN *et al.*, 2017) and create hierarchical structures. Hierarchical in the sense that a tree has multiple levels and a notion of ancestor and descendant nodes. This structure is, in fact, a directed acyclic graph, i.e., a tree in graph theory (WEST *et al.*, 1996). Henceforth, we use the terms DTs and trees interchangeably.

Due to this structural characteristic, DTs can be easily interpreted concerning how they make decisions. This is an important characteristic and advantage of these models against black-box decision models, e.g., artificial neural networks, which cannot be easily interpreted (BISHOP, 2006; GAMA, 2010). As an example, Figure 1 presents a hypothetical regression tree that predicts the chance of rain precipitation given climate descriptions. The decision process is clearly depicted in the tree structure.

Figure 1 – Hypothetical example of DT that predicts the chance of rain precipitation.



Source: Elaborated by the author.

The following statements come from classical ML literature (BREIMAN, 2001; BISHOP, 2006; RUSSELL; NORVIG, 2016). The root is the shallowest node in a DT, from which all other nodes descend. Nodes without descendants are usually referred to as terminal, decision, or leaf nodes. The remaining nodes can be referred to as inner nodes. Both the root and the inner nodes are grouped under the name of test nodes. The immediate ancestor of a node is referred to as its

parent node. Typically, a decision is made by choosing a path from the root to a leaf node. When querying a DT model about an incoming instance, we can say that the instance is sorted to the leaf (GAMA, 2010; BREIMAN *et al.*, 2017). Again, more than one path can be followed in a tree to produce a prediction (KOHAVI; KUNZ, 1997; PFAHRINGER; HOLMES; KIRKBY, 2007), but this is not the most common approach.

A path in a DT can also be seen as a DR. For this reason, DR induction algorithms are conceptually very similar to algorithms for the induction of DTs (IKONOMOVSKA; GAMA; DŽEROSKI, 2011a; IKONOMOVSKA; GAMA; DŽEROSKI, 2015; DUARTE; GAMA, 2015; DUARTE; GAMA; BIFET, 2016), though their construction differs. Following the regression DT literature, we can categorize regression tree models into two groups, according to their prediction strategy: regression and model trees (TORGO, 1997; IKONOMOVSKA; GAMA; DŽEROSKI, 2011b; IKONOMOVSKA; GAMA; DŽEROSKI, 2015). Regression trees, such as the classic Classification and Regression Tree (CART) (BREIMAN *et al.*, 2017), produce the sample mean of the target variable at the leaves, i.e., a constant value. On the other hand, model trees use a more sophisticated functional model in each leaf, i.e., the response for each instance sorted to a leaf depends on its feature values. If we consider linear model trees, they can be seen as a form of locally weighted regression, while regression trees are piecewise-constant regression models (TORGO, 1997; LOH, 2011).

Hoeffding Trees (HT) are the most popular family of tree-based OML algorithms (DOMINGOS; HULTEN, 2000; IKONOMOVSKA; GAMA; DŽEROSKI, 2011b). They receive this name because they rely on the statistical measure proposed by Hoeffding (1963) to perform split decisions, i.e., the Hoeffding Bound (HB). Hoeffding Tree Regressors (HTR) are also widely applied, mainly when instantiated as model trees (IKONOMOVSKA; GAMA; DŽEROSKI, 2011b; IKONOMOVSKA; GAMA; DŽEROSKI, 2015; OSOJNIK; PANOV; DŽEROSKI, 2017). Many incremental DR algorithms also rely on the HB in their construction. The same happens when we consider ensemble algorithms (BREIMAN, 1996; BREIMAN, 2001; GEURTS; ERNST; WEHENKEL, 2006). HTs are the most popular choice in the creation of incremental ensembles (GOMES *et al.*, 2017; KRAWCZYK *et al.*, 2017). The main idea behind creating ensemble algorithms is to join the predictions of multiple base models to obtain a combined performance higher than the one from an individual model. There are varied strategies for composing ensembles in both batch and incremental scenarios. Gomes *et al.* (2017) and Krawczyk *et al.* (2017) categorize ensemble algorithms according to the type of constituent base models, how the models are trained, and how their predictions are combined, among other aspects.

1.2 Research questions and hypotheses

By surveying the available literature, we identified that most existing online tree regressors and ensembles thereof were too computationally costly for real-world applications. Besides,

in the specific case of sliding window-based k-NN-based algorithms, we realized that the existing solutions perform a complete scan of the data buffer elements, which might become impractical in real-world solutions. We proposed the following research questions to guide our research during the development of the thesis:

Q1: Can the computational costs of Hoeffding Tree regressors be reduced without significant impacts on predictive performance?

Q2: Can efficient online tree-based ensembles be created while keeping competitive predictive performance to state-of-the-art solutions?

Q3: Is there an efficient and alternative strategy to perform nearest neighbor search queries on a data buffer that is constantly updated using a first-in, first-out data ingestion policy?

Given the research questions and the literature review, we formulated the following hypotheses to pursue further:

(Hyp. 1) *The usage of summarization and data sampling techniques can lead to a significative reduction in the computation costs for building incremental trees and decision rule-based regressors.*

(Hyp. 2) *Efficient and incremental graph-based search structures can be created to perform nearest neighbor search queries using arbitrary distance measures.*

In the next chapters of this thesis, we show how we used **Hyp. 1** to answer **Q1** and **Q2**, and **Hyp. 2** as the starting point to address **Q3**.

1.3 Main contributions

The initial plan for the thesis theme was to investigate multi-target regression problems in an OML setting and to develop more efficient decision trees and ensembles thereof for that end. However, by investigating the literature, we realized that, in general, online decision trees and decision rule solutions for regression needed further investigation.

Hence, we decided to broaden the scope of the thesis by pursuing strategies to decrease the computation costs of trees and DR-based regressors in general, including both the single-target and multi-target cases. Toward the end of the thesis development, we also investigated ways of creating more efficient algorithms for another popular family of OML algorithms, namely, incremental k-Nearest Neighbors (k-NN). Following, we list the main contributions achieved during the development of this thesis:

- We developed strategies to speed up Hoeffding Tree regression construction and save memory costs while also keeping competitive predictive performance (MASTELINI; CARVALHO, 2020; MASTELINI; CARVALHO, 2021; MASTELINI *et al.*, 2021);
- We proposed a novel tree-based ensemble regression algorithm that leverages, among other aspects, sub-bagging to significantly speed up the training step and reduce the memory costs while also boosting predictive performance (MASTELINI *et al.*, 2022);
- We proposed an algorithm to efficiently perform nearest neighbor search queries in a sliding window while also being able to handle frequent element addition and removal;
- In collaboration with researchers from multiple countries, the thesis author created the River (MONTIEL *et al.*, 2021) Python library for OML. River is one of the most popular tools for researchers and practitioners who want to develop OML solutions.

1.4 Thesis organization

This thesis is organized as a collection of papers. The first part, Chapters 2-6, comprehends articles published in the realm of creating efficient tree and decision rule-based regressors, which was the main goal pursued most of the thesis development. The first part of the thesis aims at answering research questions **Q1** and **Q2** by exploring techniques and strategies rooted at **Hyp. 1**.

The second part, Chapter 7, is related to the research internship developed at the University of Porto, where the main focus was online nearest neighbor search. This last part is more general in the sense that there is no specific focus on a single learning task. The developed solution can be applied to classification, regression, anomaly detection, and clustering, among others. This second part of the thesis aims to answer the research question **Q3**, using **Hyp. 2**.

All the chapters, except for the Chapter 8, which presents the conclusion and final remarks, are independent and self-contained, i.e., they contain all the background and references necessary for their comprehension. The first and second parts can be read independently. However, we suggest reading the chapters of the first part in the order they are presented. Although the constituent papers are independent, they solve some open issues and challenges that the preceding works might identify. Therefore, the presented order, which is chronological, also corresponds to a logical reading order. Below, we describe each chapter and how they are linked.

1.4.1 Chapter 2

Title: *"On Ensemble Techniques for Data Stream Regression"*. This chapter corresponds to an article written in collaboration with Prof. Dr. Heitor M. Gomes, Dr. Jacob Montiel, Prof. Dr. Bernhard Pfahringer, and Prof. Dr. Albert Bifet, which are researchers and former researchers

from the University of Waikato - New Zealand. The paper was published in the 2020 International Joint Conference on Neural Networks (IJCNN) (GOMES *et al.*, 2020).

We present a comparative study evaluating popular choices for creating online ensembles for regression tasks. The published paper introduces Streaming Random Patches Regressor (SRP-Reg), an ensemble algorithm that combines random subspaces and online bagging to create diverse and high-performance regressors. The main contributions of the chapter are:

- We compare multiple strategies for prediction aggregation, handling concept drift, and inducing diversity in incremental ensembles;
- We show that simple, non-reactive, concept drift adaptation strategies can be as efficient as the reactive ones given proper parameter choice;
- We empirically observe that combining median aggregation and model trees is beneficial for predictive performance;
- We also conclude that using random patches is less beneficial for regression tasks than for classification.

1.4.2 Chapter 3

Title: *2CS: Correlation-guided Split Candidate Selection in Hoeffding Tree Regressors*. After the findings reported in Chapter 2 and the experience of running and analyzing the empirical experiments of that paper, it became clear that the computational costs were an inherent problem in HTRs. An initial attempt to reduce HTR training time was presented in a paper published at the 2020 Brazilian Conference on Intelligent Systems (BRACIS), whose content is presented in this chapter (MASTELINI; CARVALHO, 2020).

In this chapter, we reduce the tree induction cost by limiting the number of features evaluated during split attempts. For such, we rely on the correlation between each feature and the target and only select the subset of features most correlated with the target for split attempts. This strategy significantly reduces the cost of training HTRs, but this comes at the cost of sometimes increasing the prediction error. One might ignore potentially useful features by selecting only a subset of features to evaluate splits on. Thus, this chapter identifies that a better research direction would be to improve the tree-splitting mechanism by itself rather than adding more heuristics on top of the already greedy-based split strategy used by the trees. The main contributions in this chapter are as follows:

- We introduce a correlation-based mechanism to speed up HTR training time that just slightly increases the memory costs of the trees;
- We evaluate the impact of the heuristic on both regression and model trees;

- We show that even by bypassing some features, the predictive performance of the HTRs is not significantly impacted, while the running time is significantly decreased;

1.4.3 Chapter 4

Title: *Using dynamical quantization to perform split attempts in online tree regressors*. Building upon the findings presented at Chapter 3, we developed an improved strategy to evaluate split attempts, which is presented in this chapter. The chapter's content was published at Pattern Recognition Letters (MASTELINI; CARVALHO, 2021).

Our proposal relies on a simple yet effective quantization technique to significantly reduce the costs of performing split attempts in HTRs. Our proposal is dubbed Quantization Observer (QO). Moreover, this chapter introduces more robust incremental variance calculation formulae used in the HTRs and decision rule regressors to calculate split merit. We show that the improved equations significantly decrease numerical approximation issues. Below, we list the main contributions of this chapter:

- We propose a novel, lightweight, and easy-to-set-up HTR split strategy that significantly decreases both the running time and memory usage;
- QO does not significantly impair the predictive performance and yields split candidates similar to those obtained via an exhaustive approach;
- The combination of QO and the improved variance calculation formulae results in reliable and efficient approximate split decisions.

1.4.4 Chapter 5

Title: *Fast and lightweight binary and multi-branch Hoeffding Tree Regressors*. We relied on a synthetic evaluation setup in the previous chapter to evaluate the split candidate evaluation algorithms. We actually isolated the tree-splitting procedure from the remaining aspects of the HTR algorithm. Hence, we were able to address the costs and performance of split evaluations directly. We extend the previous work in this chapter by applying QO to HTRs. We also go a step further and leverage the inherent characteristics of our quantization split evaluations to create multi-branch trees even for numerical features. The contents of this chapter were published in the IncrLearn workshop of the 21st IEEE International Conference on Data Mining (ICDM) (MASTELINI *et al.*, 2021), in collaboration with Dr. Jacob Montiel, Prof. Dr. Heitor M. Gomes., Prof. Dr. Bernhard Pfahringer, and Prof. Dr. Albert Bifet, from the University of Waikato.

The main contributions of the chapter as summarized as follows:

- We apply QO to HTRs in a diverse and extensive evaluation setup;

- We show that HTRs powered by our quantization-based splits are significantly faster than the original ones, spend a fraction of the memory usage, and have comparative predictive performance;

- We also show that the multi-branch trees are more efficient than the strictly binary ones while also significantly shallower, albeit wider. This last characteristic could be potentially beneficial for interpretation purposes.

1.4.5 Chapter 6

Title: *Online Extra Trees Regressor*. The works presented in Chapter 4 and Chapter 5 enable building HTRs efficiently and with reduced memory costs. Although HTR ensembles powered by the quantization-based splits are viable options that ought to be faster than the original ones, HTR forests' costs might still become prohibitive in real-world applications. For that reason, and inspired by a batch-based algorithm, we introduce Online Extra Trees (OXT). OXT utilize random split points in their modified HTRs, which are substantially faster to calculate. Our proposed forest algorithm also relies on sub-sampling, i.e., sub-bagging, to train the trees, which significantly decreases training time with no noticeable impact on predictive performance. This chapter presents the contents of the paper published at IEEE Transactions on Neural Networks and Learning Systems (MASTELINI *et al.*, 2022), in collaboration with Dr. Felipe K. Nakano and Prof. Dr. Celine Vens, from the KU Leuven University - Belgium.

The main contributions of the chapter are listed as follows:

- We introduce a novel incremental tree-based ensemble regressor, which is generally more accurate than the previous state-of-the-art solutions while being significantly faster and using significantly fewer memory resources;

- We create a random-based split point selection procedure that has a constant asymptotic time complexity, a superior result to the one presented at Chapter 4, albeit relying on the same variance calculation formulae introduced in that chapter;

- We introduce the usage of sub-sampling to create ensembles of HTs, leveraging the inherent stability of these tree-based models, as described by Gomes *et al.* (2021);

1.4.6 Chapter 7

Title: *SWINN: efficient nearest neighbor search in sliding windows using graphs*. Shifting the focus from DTs and decision rules, we decided to explore the nearest neighbor search during a research internship developed at the University of Porto under the supervision of Prof. Dr. João Gama. The limitations of online k-NN algorithms in terms of computational costs were a constant discussion brought by researchers and practitioners using the River (MONTIEL *et*

al., 2021) library. Upon proper inspection of the literature, we realized a need for more efficient solutions to perform k-NN incrementally, mainly when considering a sliding window-based buffer. Hence, we proposed Sliding Window-based Incremental Nearest Neighbors (SWINN) to tackle this issue. SWINN is significantly faster than the traditional approach and can work with arbitrary distance measures. The resulting paper, written in collaboration with Prof. Dr. Bruno Veloso, Dr. Max Halford, and Prof. Dr. Gama, and presented in this chapter, was sent for review in Information Fusion (Elsevier). The paper received minor change requests from the reviewers and is currently being prepared for resubmission.

The main contributions of the chapter are listed next:

- We introduce SWINN, a graph-based search index that can significantly speed up the nearest neighbor search when compared to a complete scan of all available data;
- Our solution works with any distance metric and supports frequent element addition and removal;
- We evaluate SWINN in a broad experimental setup and assess how each hyperparameter of our proposal affects its final performance;
- We evaluate two case studies to assess the potential of SWINN in realistic situations compared with state-of-the-art online ML algorithms.

1.4.7 Chapter 8

In this chapter, we present this thesis' conclusions and discuss the limitations of our research, open challenges, and possible future studies.

1.4.8 Appendix

We present the supplementary material of Chapter 7 in Appendix A. We also list additional articles published during the Ph.D. development period in Appendix B.

CHAPTER

2

ON ENSEMBLE TECHNIQUES FOR DATA STREAM REGRESSION

Publication information: This chapter is an article published at the IEEE International Joint Conference on Neural Networks.

Reference: GOMES, Heitor Murilo et al. On ensemble techniques for data stream regression. In: **2020 International Joint Conference on Neural Networks (IJCNN)**. © IEEE, 2020. p. 1-8. Reprinted, with permission, from the authors.

2.1 Abstract

An ensemble of learners tends to exceed the predictive performance of individual learners. This approach has been explored for both batch and online learning. Ensembles methods applied to data stream classification were thoroughly investigated over the years, while their regression counterparts received less attention in comparison. In this work, we discuss and analyze several techniques for generating, aggregating, and updating ensembles of regressors for evolving data streams. We investigate the impact of different strategies for inducing diversity into the ensemble by randomizing the input data (resampling, random subspaces and random patches). On top of that, we devote particular attention to techniques that adapt the ensemble model in response to concept drifts, including adaptive window approaches, fixed periodical resets and randomly

*<http://www.ieee.org/publications_standards/publications/rights/rights_link.html>

determined windows. Extensive empirical experiments show that simple techniques can obtain similar predictive performance to sophisticated algorithms that rely on reactive adaptation (i.e., concept drift detection and recovery).

2.2 Introduction

The application of machine learning to data streams has grown in importance in recent years due to a large amount of real-time data generated by networks, mobile phones, and the wide variety of sensors currently available. Building predictive models from data streams are central to many applications. One example is the Internet of Things (IoT) applications, where connected sensors yield a large amount of data in short periods. To build predictive models from streaming data, we need to either settle for traditional offline learning or employ algorithms capable of learning incrementally. A significant setback with the offline learning approach is that it is slow to react to changes in the domain, and these changes can have a catastrophic impact on the predictive model performance since the patterns in which the model was trained on are no longer valid.

Often, the application of a single decision model may lead to subpar performance in online scenarios, given the previously mentioned challenges. As a consequence, algorithms that combine several models, i.e. ensemble methods, are a trend for supervised learning for both static and streaming data. Ensembles enable leveraging the power of multiple learners towards the same goal, whereas alleviating their individual limitations.

Ensemble learning has been thoroughly investigated for data stream classification (GOMES *et al.*, 2017). Consequently, several methods were proposed (OZA; RUSSELL, 2001a; GOMES *et al.*, 2017) to this end. Recently, more methods were proposed for data stream regression, such as the Adaptive Random Forest Regressor (ARF-Reg) (GOMES *et al.*, 2018). There are at least three relevant aspects to be considered when proposing an ensemble learner, either for regression or classification: combination, generation, and the update dynamics. The combination (or voting) strategy describes how the individual predictions are aggregated to obtain the ensemble prediction. For classification, a common method is majority vote, while for regression, the mean is commonly used. The generation method defines how the base models are trained, commonly including some mechanism to enforce diversity among the base learners. A traditional approach is to train learners on different subsets of instances (e.g., Bagging (BREIMAN, 1996)), features (e.g., the Random Subspaces Method (HO, 1995)) or both (e.g., Random Patches (HO, 1995) and Random Forests (BREIMAN, 2001)). The update dynamics is fundamental when dealing with streaming data, specifically evolving data streams, as it defines how (and when) base models will be reset or updated to reflect changes to the underlying data distribution.

These three aspects of ensemble learning were thoroughly investigated for data stream classification. For example, the Adaptive Random Forest (ARF) for classification algorithm (GOMES

et al., 2017) includes an empirical comparison between majority vote against a weighted majority vote. Conversely, the regression version of ARF, namely ARF-Reg (GOMES *et al.*, 2018), presents only results considering a simple linear combination (the mean) of the predictions. Therefore, there is room for the investigation of other combination techniques, the impact of different reset strategies to deal with concept drifts, and how to generate diverse learners for a regression problem.

The main contributions of this work are the following:

- We discuss and analyze several techniques to train base learners, combine their predictions and update them actively (or reactively) to address concept drifts.
- We benchmark existing algorithms and the proposed ensemble variants using 17 datasets. This lead us to insightful conclusions, such as the high performance obtained by relatively simple models (e.g., k Nearest Neighbors) in comparison to ensemble models that implement complex drift detection and recovery.
- We provide empirical evidence that indicate that using reactive strategies to adapt to concept drifts might not be necessary given an ensemble where base learners are trained using windows of varying sizes, thus reset at different time intervals.

The remainder of this work is organized as follows. In section 2.3, we review ensemble methods for data stream regression and classification. Section 2.4 contains the description of strategies to build, combine and update ensemble models designed for evolving data stream regression. Section 2.5, introduces the four sets of experiments that provide an insightful analysis of the presented ensemble strategies. Finally, in section 2.6 we present our concluding remarks and directions for future work.

2.3 Related work

The **Fast and Incremental Model Trees (FIMT-DD) (FIMT-DD)** (IKONOMOVSKA; GAMA; DŽEROSKI, 2011b) is the most widely used algorithm to build incremental regression trees for streaming data. Similarly to the Hoeffding Tree algorithm (DOMINGOS; HULTEN, 2000), FIMT-DD starts with an empty tree that keeps statistics from arriving data until a grace period is reached. The features are ranked according to their variance w.r.t the target variable to decide for splits, and if the two best-ranked differ by at least the Hoeffding Bound (HOEFFDING, 1963), the node splits. FIMT-DD includes a change detection scheme that periodically flags and adapts subbranches of the tree where significant variance increases are observed. In Ikonomovska *et al.* (2011), the authors propose the **On-line Regression Trees with Options (ORTO)** algorithm, that introduces 'option' nodes, which allow an instance to follow all the branches available in a tree node. The **Adaptive Model Rules (AMRules)** (ALMEIDA;

FERREIRA; GAMA, 2013) learns both an ordered and an unordered rule set from a data stream. To detect and adapt to concept drifts, each rule is associated with a Page-Hinkley drift detector (MOUSS *et al.*, 2004), which prunes the rule set given changes in the incoming data.

The development of ensembles of regressors attracted less attention than ensembles of classifiers for streaming data, even though there is a sizeable amount of literature on this topic for batch learning (MENDES-MOREIRA *et al.*, 2012). For streaming data, Ikonomovska, Gama and Džeroski (2015) proposed the **online random forest (ORF)** and **online bagging (OBag)** ensembles that use the FIMT-DD as the base learner. Based on empirical experiments, the authors concluded that the ORTO-A (online option trees with averaging) outperformed both OBag and ORF in terms of Mean Squared Error (MSE). More recently, the **Adaptive Random Forest regressor (ARF-Reg)** (GOMES *et al.*, 2018), an adaptation of the data stream classifier (GOMES *et al.*, 2017) of the same name was proposed. ARF-Reg builds a forest of FIMT-DD trees as ORF, the main difference between both algorithms is that ARF-Reg employs one instance of the Adaptive WINdow (ADWIN) algorithm (BIFET; GAVALDA, 2007) per tree to detect concept drifts. The way in which randomization is added during model generation in a random forest is particular to decision trees. A more general approach is to use the random subspaces method (HO, 1995) as in **Heuristic Updatable Weighted Random Subspaces (HUWRS)** (HOENS; CHAWLA; POLIKAR, 2011) and **Streaming Random Patches (SRP)** (GOMES; READ; BIFET, 2019) algorithms. SRP trains each base learner on a subset of features and instances from the original data, namely a random patch (LOUPPE; GEURTS, 2012). This strategy to enforce diverse base models is similar to the one in the random forest, yet it is not restricted to using decision trees as base learner. Moreover, in Gomes, Read and Bifet (2019) the overall results (in terms of accuracy) for SRP outperformed the adaptive random forest (ARF) (GOMES *et al.*, 2017) in a multitude of datasets.

In this work, we introduce techniques that can be applied to ensembles of regressors for streaming data. We focus on four aspects: combination, generation, base learner, and reset strategies. For generation, we explore techniques that induce a diverse set of base models by training them on different subsets of instances (bagging), features (random subspaces) or both (random patches). To combine the predictions, we investigate the benefits of using the median instead of the mean. We also investigate the impact of different base learners (incremental trees and k-nearest neighbors) or variations of these (i.e., adapting the leaves of the tree). Finally, we challenge the well-established strategy of resetting base learners according to some drift detection algorithm (GOMES *et al.*, 2017; GOMES *et al.*, 2018; GOMES; READ; BIFET, 2019) against simpler strategies, such as fixed windows of varying sizes.

2.4 Ensemble strategies for data stream regression

In simple terms, an ensemble learner is a set of base models and an integration method to combine their predictions. When applied to stream data, it may also incorporate some reset dynamics to adapt the ensemble to potential changes in the data distribution. There is a vast literature concerning strategies to improve ensemble classifiers for streaming data (GOMES *et al.*, 2017), yet not as many approaches have been thoroughly investigated for ensembles of regressors. In this section, we discuss strategies that can potentially leverage an ensemble learner for data stream regression.

2.4.1 Generation - Training and Diversity Induction

There are different approaches for generating (training) base models. The motivation for the development of such strategies is to enforce diversity into the ensemble. If all the base models make homogeneous predictions, it is clear that their combination is no better than just using one of them. Many algorithms provide mechanisms to induce diversity implicitly by training each base model on different subsets of the data. Canonical examples include bagging (BREIMAN, 1996), the random subspaces method (HO, 1995), random forests (BREIMAN, 2001), and random patches (LOUPPE; GEURTS, 2012). These techniques were successfully adapted and applied to data stream classification (OZA; RUSSELL, 2001a; HOENS; CHAWLA; POLIKAR, 2011; GOMES *et al.*, 2017; GOMES; READ; BIFET, 2019), and some of them to regression (IKONOMOVSKA; GAMA; DŽEROSKI, 2015; GOMES *et al.*, 2018). We employ generation techniques that do not rely on a specific algorithm as the base learner. Precisely, we explore Random Subspaces (RS), Bagging (BAG)[2], and Random Patches (RP). These generation techniques enforce that each base model is trained with different subsets of instances (BAG), features (RS) or both (RP). On top of that, these are techniques that modify the training data presented to each base learner without explicitly changing the base learner algorithm (e.g., random forests manipulate the individual tree construction algorithm). A similar approach is explored in Gomes, Read and Bifet (2019) for data stream classification.

2.4.2 Combination

We analyze two strategies for aggregating the predictions, the mean and the median. Aggregating the ensemble prediction as the mean of its members' predictions is a simple and often effective strategy used in a multitude of algorithms (MENDES-MOREIRA *et al.*, 2012). One drawback of the mean as a measure of central tendency is that it can be affected by any single value that is either too high or too low in comparison to the rest of the sample. In our context, it is desirable to avoid situations where a single model prediction can have a potentially harmful

[2]We refer to it simply as BAG. However, we are in fact using the resampling strategy introduced in (BIFET; HOLMES; PFAHRINGER, 2010) where Poisson($\lambda = 6$) is used instead of Poisson($\lambda = 1$), as in the original Online Bagging adaptation by Oza (OZA; RUSSELL, 2001a).

influence on the overall prediction. For example, after resetting a base model, the relatively 'new' model may produce predictions that deviate too far from others, simply because it has not been trained on a sufficiently large amount of data. However, the other way around can be true as well, i.e., a single learner's prediction positively influence the overall combined vote. To investigate further we propose comparing the mean aggregation against a median aggregation. Using the median to aggregate ensemble predictions was used in the Bragging (bootstrap robust aggregating) algorithm, a variant of Bagging proposed by Bühlmann in (BÜHLMANN, 2003).

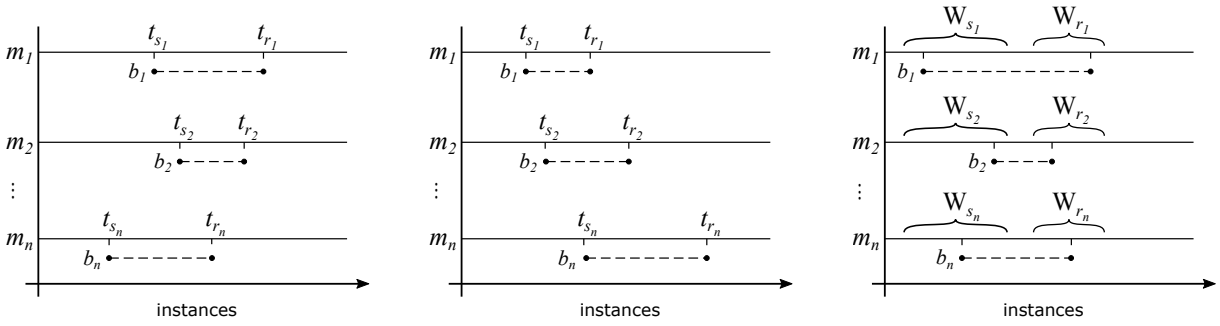
2.4.3 Reset Strategy

The reset strategy in an ensemble, designed to cope with evolving data streams, is an utterly important component. The strategies for maintaining learning algorithms up-to-date even when faced with concept drifts are often categorized as explicit (or reactive) and implicit (or proactive). Reactive methods rely on an algorithm (i.e., drift detector) that indicates the need to reset the base models, while active methods constantly resets the base models according to some predefined strategy. Reactive strategies are popular and often employed alongside ensemble methods for data stream classification (GOMES *et al.*, 2017; BIFET; GAVALDÀ, 2009) and regression (IKONOMOVSKA; GAMA; DŽEROSKI, 2011b; GOMES *et al.*, 2018).

We now describe three strategies for resetting the ensemble model, depicted in Figure 2. All of them follow the same principles of replacing an active base model with a model that has been trained without influencing the ensemble decisions (namely, a background model). The intuition behind training a model before adding it to the ensemble is to avoid an underfitted model interfering with the ensemble predictions. The differences among these strategies lie on how they determine the start of the training for the background model (t_s) and the replacement (t_r) of the current model.

- **Adaptive Window.** A drift detection algorithm monitors the error of each base learner, and whenever a drift is signaled the associated base learner is reset (thus ending its training window);
- **Fixed Window.** The length of each training window is predefined, such that each model is reset after reaching the maximum number of instances.
- **Random Window.** This approach adds another level of randomization by determining the window length of each learner randomly.

The **adaptive window** depends on the change detection algorithm used to identify potential drifts (warnings) and actual drifts. We use the ADaptive WINdow (ADWIN) (BIFET; GAVALDA, 2007) algorithm, but other detectors could be used as well. The error of each base model is monitored by a different instance of ADWIN. A warning signal determines when to



(a) Adaptive Window. (b) Fixed Window. (c) Random Window.

Figure 2 – Reset model strategies. Let m_i denote the i-th model in the ensemble, in all strategies a background model b_i starts training at t_{s_i} and it replaces m_i at time t_{r_i}. After a model is replaced, the reset process starts again. In Fig. 2a the ADWIN drift detector is used, t_{s_i} and t_{r_i} are set according to changes in the performance of a model i as detected by ADWIN. In Fig. 2b a fixed-size window is used between t_{s_i} and t_{r_i}. We vary the size of such window for each member i of the ensemble, to ensure that background models are trained on different number of instances and to avoid replacing all members of the ensemble at the same time. In Fig. 2c two adjacent windows W_{s_i}, W_{r_i} are defined. These windows correspond to the range in which the training of the background model starts (t_{s_i}) and when the model replacement takes place (t_{r_i}). Both events happen at random within the corresponding window.

start training the background model (t_s), and a drift signal determines when to replace the model (t_r). Effectively, t_s and t_r are automatically set, but the detection algorithm itself adds some hyperparameters (i.e., the confidence λ for ADWIN), thus it is not entirely automatic. This strategy has been explored in multiple works for both regression (GOMES *et al.*, 2018) and classification (GOMES *et al.*, 2017; GOMES; READ; BIFET, 2019).

In the **fixed window** strategy, we avoid simultaneously resetting the base models by adding a small shift to the hyperparameters t_s and t_r, such that the training of the background model and the replacement of the current model are slightly different for each base model. This way, we also avoid replacing all members of the ensemble at the same time, which effectively represents resetting the ensemble. We can associate t_s and t_r with the warning and drift detection from the adaptive window. This strategy is similar to how data is buffered to sequentially train batch learners in the Fast and Slow Framework (MONTIEL *et al.*, 2018), where batch and stream learning methods operate together.

To generate **random windows** that are neither too short nor too large, we constraint the length of the window l to be a positive integer approximately in the interval $l \in (t_0 - 1/W, t_0 + 1/W)$. Given hyperparameters t_0 and W, the probability that the window of training ends after observing t instances is given by Eq. 2.1.

$$Pr[reset] = 1/(1 + e^{-4(t-t_0)/W}) \tag{2.1}$$

As the number of instances observed t approaches $t_0 - W/2$, the probability that a reset is triggered starts increasing, reaching $Pr[reset] = 0.5$ when $t = t_0$, and approaching $Pr[reset] = 1.0$

after $t_0 + W/2$. Other similar approaches, such as uniformly choosing a random number between $t_0 - W/2$ and $t_0 + W/2$, could be used, but we remark that they would produce similar results.

The question that we want to answer with both the fixed and random window approaches is:

Does signaling when to reset base models using an accurate drift detector positively influences the predictive performance of the ensemble, OR the positive effects are caused by periodically resetting the base learners?

In Bifet (2017), the authors presented a similar empirical study to verify the relevance that a drift detector plays on a classification system. The conclusions obtained by the authors were that a fine-tuned fixed window length was able to overcome a system that relies on an adaptive window length (drift detector). The caveat is that it is not trivial to determine the window length ahead of time in an optimal way. However, one may want to avoid too short or too large (or infinite) window lengths. Windows that are too short do not allow the base model to learn any concept effectively, resulting in suboptimal performance. Conversely, windows that are too large imply the risk of keeping old concepts within the ensemble.

2.4.4 Base learner

We experiment with two popular regression algorithms as the base learners for the ensemble variations. The first is the Hoeffding Tree Regressor (HTR), which is a variation to its classifier counter-part by Domingos and Hulten (DOMINGOS; HULTEN, 2000). Similarly to FIMT-DD (IKONOMOVSKA; GAMA; DŽEROSKI, 2011b), HTR split decisions are based on the variance information, and the aggregation at the leaves can either be performed by a linear model (i.e., a perceptron) or the mean target values of examples reaching the leaf. Nonetheless, HTR does not include mechanisms for concept drift adaptation as FIMT-DD. This fact, though making HTR not coping with non-stationary distributions when working standalone, shall improve its computation resource usage over FIMT-DD. The second algorithm is k nearest neighbors (KNN). KNN is a common baseline for both classification and regression. The basic kNN regressor searches for the k instances that are closer (w.r.t. a given distance metric) to an instance whose target value has to be predicted. The predicted value is the unweighted mean of the k nearest instances found. KNN is known to be a stable learner, i.e., given a small variation in the training sample for two KNN models, their predictions will be fairly similar. There are different approaches to enforce instability to KNN models, such as injecting randomness to the distance metrics (ZHOU; YU, 2005) or using different random subspaces to build the training samples (HO, 1998).

It was observed in Ikonomovska, Gama and Džeroski (2015) and Gomes, Read and Bifet (2019) that as Hoeffding trees grow larger they become more similar w.r.t. their predictions. We shall observe a similar behavior if KNN is employed with bagging. In this work, we refrain

from applying adaptations to the base models with the sole intention of inducing diversity as we lack the space for appropriate analysis and discussion of such an important topic. Furthermore, we rely on ensemble generation strategies to enforce diverse predictions for KNN and HTR, precisely by using random subspaces and random patches.

2.5 Experiments

For every experiment we apply a test-then-train evaluation strategy, i.e., each instance is used for testing and then used for training. We analyze how the learning algorithms performs in terms of Root Mean Square Error (RMSE) in different scenarios including real and synthetic data. There are 17 datasets used in the experiments, including real (7) and synthetic datasets (10), such as Hyperplane (Hyper) and Radial Basis Function (RBF) variations. We use three variations of Hyper and RBF synthetic datasets, each of them simulating a different type of drift. Synthetic datasets variants identified with (G) and (A) simulates gradual and abrupt drifts every $125K$ instances (i.e., $125K$, $250K$ and $375K$). The window of change for abrupt drifts is 1, and $20,000$ for gradual drifts. Variants (I) simulate incremental concept drifts. The summary statistics of the datasets are shown in Table 1.

Table 1 – Characteristics of the evaluated datasets. Simulated Drifts: (A) Abrupt, (G) Gradual, (I) Incremental. The first group (top) contains the real-world datasets, and the second group (bottom), the synthetic datasets.

Dataset	#Instances	#Numeric features	#Categorical features
Abalone	4977	7	1
Bike	17379	12	0
CalHousing	20500	8	0
House8L	22784	8	0
House16H	22784	16	0
MetroTraffic	48204	4	3
Pol	15600	48	0
Ailerons	13750	40	0
Elevators	16599	18	0
Fried	40768	10	0
MVDelve	40967	7	3
Hyper(A)	500000	10	0
Hyper(G)	500000	10	0
Hyper(I)	500000	10	0
RBF(A)	500000	20	0
RBF(G)	500000	20	0
RBF(I)	500000	20	0

Some hyperparameters were fixed throughout all experiments. The ensemble variants (e.g., ARF-Reg) were executed with 30 base learners and a subspace size of 60%. Most of the

base learners are based on variants of a Hoeffding Tree, including the HTR and FIMT-DD. In these cases, we used a configuration that has been successful for ensemble classifiers, i.e., a grace period of 50 and a higher confidence ($\delta = 0.01$). BAG, RP, and RS, their variants and base learners, were executed and implemented[3] in *scikit-multiflow* (MONTIEL *et al.*, 2018)[4]. The remaining considered algorithms are available in *MOA* (BIFET *et al.*, 2018). Unless otherwise indicated, all the algorithms were performed using their standard hyperparameters, according to their implementations.

To facilitate the identification of the ensemble variants, we introduce the following naming convention. **Generation:** random patches (RP), bagging (BAG), random subspaces (RS); **Combination:** mean (μ), median (med); **Reset strategy:** adaptive window (a), fixed window (f), random window (r); **Base learner:** Hoeffding Tree regressor with mean leaves (HTR_m); Hoeffding Tree regressor with perceptron leaves (HTR_p); k-Nearest Neighbors regressor with mean aggregation (KNN). For example, when identifying a variant of Random Patches using Mean aggregation, Adaptive Window reset strategy, and HTR with perceptron at the leaves, we write RP^{a}_{μ}-HTR_p.

Our goal is to present and discuss the impact of the strategies discussed in section 2.4, as well as compare some of them against existing algorithms. It is infeasible to report all possible configurations due to the large number of combinations. Therefore, we organize the experiments in four groups to balance a breadth analysis with an in-depth analysis. In the first set of experiments we analyze the impact of the Generation and Combination strategies. We follow that experiment with an analysis of the impact of the base learner to some of the ensemble variations and how they compare against single instances of the base leaner algorithms. The third experiment was designed to answer the question posed in section 2.4, which challenged the importance of hybrid solutions that combined drift detectors to the ensemble (i.e., adaptive window reset strategies). Finally, the last set of experiments compare some of the ensemble variants against algorithms from the literature.

2.5.1 Generation and Combination

To verify the impact of both the generation and combination strategies we present the experiments in Table 2. For this analysis, we fixed the base learner as HTR_m. We observe that the BAG variants obtain the best results overall. Therefore, for the given datasets, it is not possible to conclude that improvements could be observed in terms of RMSE when employing random subspaces as part of the generation process. One possible cause can be that the majority of the datasets contain mostly relevant features, i.e., by building models on subsets of the feature set, it is not possible to obtain reasonably accurate models, which in turn negatively impacts the aggregated predictions. In general, the median combination approach was not more accurate than

[3] https://github.com/jacobmontiel/StreamingRandomPatchesRegressor

[4] The main content from scikit-multiflow, including the contributions of this chapter, have since been ported to the River library.

the mean, which shows that even though the mean is a less stable measure of central tendency, it does not seem to influence the ensemble performance negatively. Therefore, the hypothesis that extreme values may lead the combination astray could not be observed in these experiments. A possible explanation is the type of base learner, i.e., HTR_m, produced predictions that are more stable than the predictions from HTR_p.

Table 2 – Generation and combination analysis (generation=[BAG | RP | RS], combination=[mean (μ) | median (med)], base learner=HTR_m), reset strategy=adaptive (a).

Dataset	BAG^a_μ	$\text{BAG}^a_{\text{med}}$	RP^a_μ	RP^a_{med}	RS^a_μ	RS^a_{med}
Abalone	2.5911	2.6346	**2.5230**	2.5794	3.0465	3.0629
Bike	**81.0924**	82.0560	92.1240	89.1398	100.9140	98.1813
CalHousing	**63000.7071**	63447.0723	66238.8810	65327.8682	72371.6388	73694.5632
House8L	**36357.7785**	37425.4647	36674.6710	37662.6646	37238.0024	38038.3599
House16H	**39807.9324**	40503.2810	40974.8801	41893.5338	41366.8500	41694.6820
MetroTraffic	**1864.3562**	1878.4567	1868.4516	1881.3474	1909.7701	1926.9327
Pol	**39.8561**	40.9659	40.0456	40.9348	42.5239	43.0442
Ailerons	**0.0000**	0.0000	0.0000	0.0000	0.0000	0.0000
Elevators	0.0048	0.0050	0.0048	0.0049	**0.0046**	0.0047
Fried	2.8504	**2.8436**	3.3891	3.4347	4.7477	5.1362
MVDelve	2.7978	**2.3556**	3.4039	2.8376	7.2215	7.4087
Hyper(A)	**4.7393**	4.7695	5.1514	5.2270	5.6156	5.7496
Hyper(G)	**4.7314**	4.7739	5.1578	5.2315	5.6322	5.7745
Hyper(I)	**73.8283**	75.1720	75.1034	75.5369	75.7321	76.3416
RBF(A)	24.3243	26.4591	**22.6989**	23.8939	29.6176	29.9555
RBF(G)	**24.5419**	26.2645	24.9022	26.5333	29.6681	29.9662
RBF(I)	29.2339	29.2461	**29.1266**	29.1732	30.4916	30.6660
Avg. rank	**1.59**	2.88	2.59	3.65	4.65	5.65
Avg. rank real	**1.29**	3.00	2.57	3.71	4.71	5.71
Avg. rank synth.	**1.80**	2.80	2.60	3.60	4.60	5.60

2.5.2 Base learners

To analyze the impact of the base learner we compare variations of BAG^a_μ with KNN, HTR_m and HTR_p. On top of that, we also present the stand-alone results for these three algorithms with the purpose of presenting a clear baseline (i.e., it is not reasonable to use an ensemble if a single model is more accurate). The results are depicted on Table 3. We highlight that KNN obtained a low RMSE for many datasets, and in overall it was on pair with BAG^a_μ-HTR_p. It was not possible to leverage the good individual results of KNN in BAG^a_μ-KNN. One possible explanation is that KNN is a stable learner and just by slightly changing the subset of instances being used by each of the models it was unable to produce better results. In fact, by comparing KNN and BAG^a_μ-KNN, they are quite similar and often the best result is in favor of KNN. When comparing BAG^a_μ-HTR_m and BAG^a_μ-HTR_p against HTR_m and HTR_p we can observe that the ensemble models were able to outperform a single base learner. In general, using the perceptron improves the performance in comparison to using the mean to aggregate the predictions at the

leaves. This fact can be observed when comparing HTR_m and HTR_p, as well as their ensemble versions.

Table 3 – Base learner analysis (generation=BAG, combination=mean (μ), learner=[KNN | HTR_m | HTR_p], reset strategy=adaptive (a)).

Dataset	KNN	HTR_m	HTR_p	BAG^a_μ-KNN	BAG^a_μ-HTR_m	BAG^a_μ-HTR_p
Abalone	**2.3264**	3.0540	2.8726	2.3268	2.5911	2.5719
Bike	**62.3067**	108.0877	85.1775	62.3440	81.0924	69.0090
CalHousing	89876.9806	85204.2071	72327.5408	90212.1338	**63000.7071**	63307.2304
House8L	51046.4284	40883.3315	40874.6389	51026.8462	36357.7785	**35860.5305**
House16H	51164.6143	43890.7514	44301.9024	51151.9541	**39807.9324**	40028.5144
MetroTraffic	1945.0847	1951.2740	1910.3972	1945.8638	1864.3562	**1858.1606**
Pol	**18.2383**	26.4236	26.5224	23.1435	39.8561	36.1823
Ailerons	**0.0000**	0.0000	0.0000	0.0000	0.0000	0.0000
Elevators	0.0069	0.0054	0.0052	0.0068	**0.0048**	0.0048
Fried	2.6746	2.7908	2.8103	**2.6731**	2.8504	2.9802
MVDelve	8.3187	3.8745	3.8762	8.4289	**2.7978**	3.6996
Hyper(A)	**3.0400**	5.9205	5.0287	9.7130	4.7393	4.5308
Hyper(G)	**3.3039**	5.9094	5.0714	9.3744	4.7314	4.5717
Hyper(I)	50.9431	79.8120	54.4025	**50.9248**	73.8283	65.4014
RBF(A)	17.9073	23.2324	20.4353	17.9242	24.3243	**14.8843**
RBF(G)	18.7157	23.1230	20.5220	18.7317	24.5419	**14.5837**
RBF(I)	29.2988	**28.0289**	28.0496	29.2921	29.2339	28.4256
Avg. rank	3.06	4.18	3.65	3.94	3.47	**2.71**
Avg. rank real	3.43	4.57	3.86	3.86	2.86	**2.43**
Avg. rank synth.	**2.80**	3.90	3.50	4.00	3.90	2.90

2.5.3 Reset strategy

We apply three techniques to continuously reset the base models, and, thus, keep the ensemble up-to-date with the latest concepts. Each of these techniques are highly influenced by their hyperparameters, which directly influence how many instances will be used for training each base model before it is reset. We experimented with three variations of hyperparameters for the fixed and random windows, alongside a version that never resets the base models (no-reset), and one that uses an ADWIN change detector (adaptive window). More details about these variations are presented below.

- **fixed ($\mathbf{f}_s$) and random ($\mathbf{r}_s$) small.** Trigger background learner creation: $t_s = 400$; Trigger replace: $t_r = 700$; random window: $W_s = W_r = 200$
- **fixed ($\mathbf{f}_m$) and random ($\mathbf{r}_m$) medium.** Trigger background learner creation: $t_s = 1500$; Trigger replace: $t_r = 2500$; random window: $W_s = W_r = 800$
- **fixed ($\mathbf{f}_l$) and random ($\mathbf{r}_l$) large.** Trigger background learner creation: $t_s = 2500$; Trigger replace: $t_r = 5000$; random window: $W_s = W_r = 2000$
- **adaptive.** Trigger background and replace according to the drift detector.

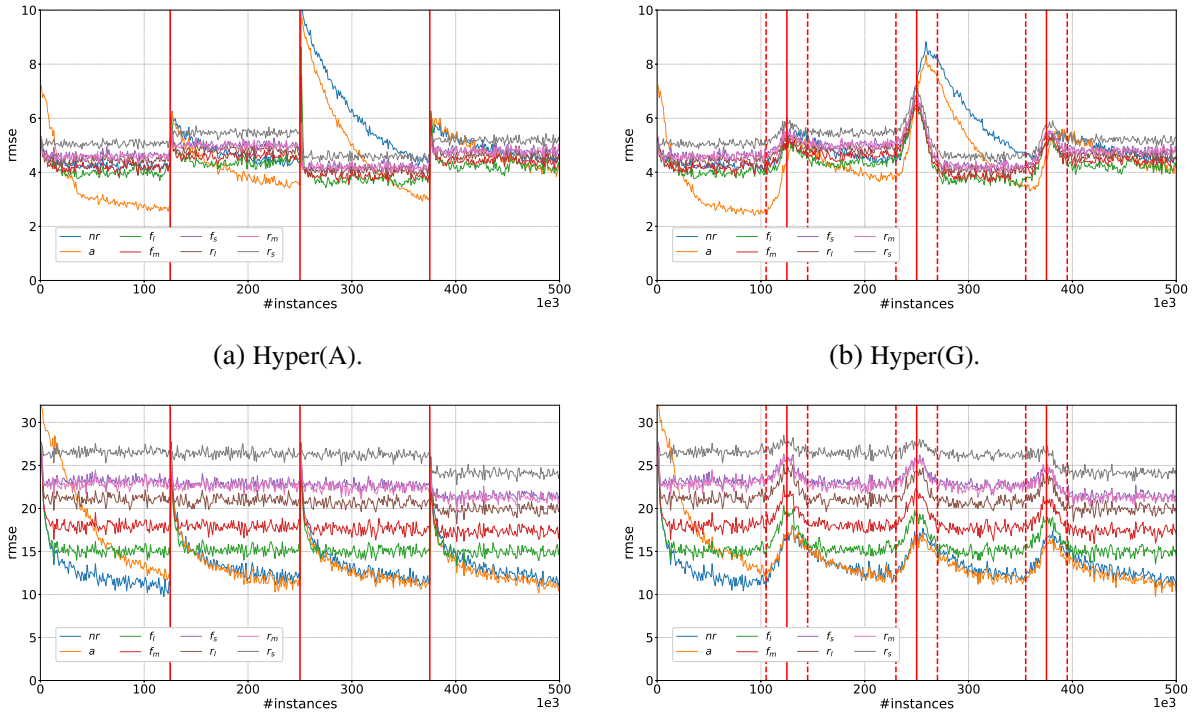
- **no-reset.** The base models are never reset.

One of the goals of the fixed and random reset strategies was to avoid resetting the base models simultaneously. The fixed window reset learners at different times and with different window sizes, the hyperparameters only define the length of the 'first' ensemble member, the others have increasing window lengths. Similarly, the random window strategy reset learners with about the same window size, but at slightly different times (depending on hyperparameter W).

Table 4 presents the results for the different reset strategies. We observe that using larger windows, for both fixed and random, improve the overall results. Analyzing the RMSE over time in Figure 2, it is noticeable that the fixed and random windows tend to adapt to concept drifts fast and without major (and long) variations to the average RMSE. This can be attributed to the fact that the base learners are reset at different times, which generates a mix of learners trained only on the latest concept and learners trained on a larger window. A counter-intuitive result is that no-reset outperforms the adaptive strategy for most of the synthetic datasets with simulated concept drifts in Table 4. Complementing the analysis with the plots from Figure 2, we can observe that the adaptive (a) variation closely resembles the no-reset results, and often recovers from concept drifts faster (Figures 3a and 3b). The ability to adapt to new concepts even if the base learners are never reset is justified by the use of no-reset with Hoeffding trees that are allowed to keep growing indefinitely. Even though this allows the trees to adapt to new concepts, it applies a heavy toll on the computational resources. The best average rankings are obtained when using a fixed window and the 'large' parametrization (f_l), which has a reasonable compromise between smaller and longer windows, i.e., a configuration in-between 'small' (f_s) and no-reset (nr).

Table 4 – Reset strategy (generation=RP, combination=median, learner=HTR$_p$, reset strategy=fixed | no-reset | random | adaptive (a)).

Dataset	f_s	f_m	f_l	no-reset	r_s	r_m	r_l	adaptive
Abalone	2.2405	2.2851	2.2804	2.2813	**2.2154**	2.2592	2.2870	2.2862
Bike	92.7782	81.9534	72.1119	**67.0228**	108.2110	100.1860	93.8778	73.4341
CalHousing	61709.5065	63947.9028	65687.8560	65809.6456	**60076.6916**	62724.3415	62055.3426	66349.6414
House8L	39343.1472	37711.8951	37184.1753	**37137.5179**	41328.1239	39208.1627	38596.7054	37718.3326
House16H	43781.4843	42412.0779	42122.2324	43091.4831	45354.9294	43583.8446	42921.1098	**41855.8577**
MetroTraffic	1811.5875	1824.2094	1842.6123	1852.6115	**1782.2871**	1811.0393	1818.1665	1859.2225
Pol	25.7078	**22.3910**	22.8605	23.1437	29.9345	25.1886	24.1786	24.8740
Ailerons	**0.0000**	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
Elevators	0.0054	0.0052	0.0051	**0.0050**	0.0054	0.0054	0.0054	0.0050
Fried	3.3059	3.0930	2.9059	**2.5176**	3.6136	3.2882	3.1752	3.0728
MVDelve	4.2186	3.3576	3.3580	4.7204	5.2824	4.1712	3.9506	**2.5276**
Hyper(A)	4.6818	4.3225	**4.1641**	5.1618	5.0834	4.6458	4.5177	4.9436
Hyper(G)	4.8392	4.4764	**4.2935**	5.1692	5.2234	4.8038	4.6746	4.9526
Hyper(I)	51.9050	**51.4357**	52.1003	67.0243	53.0485	51.6913	51.6470	67.2361
RBF(A)	22.6324	17.8684	15.4194	**13.5661**	25.8346	22.3086	20.7128	14.8983
RBF(G)	22.9855	18.3157	15.7911	**13.6283**	26.0139	22.6718	21.1338	15.2901
RBF(I)	28.8979	28.7884	28.7029	**28.3937**	29.0184	28.8918	28.8548	28.5633
Avg. rank	5.12	3.29	**3.00**	3.94	6.29	5.12	4.76	4.47
Avg. rank real	4.71	3.86	**3.43**	4.14	5.00	4.86	4.86	5.14
Avg. rank synth.	5.40	2.90	**2.70**	3.80	7.20	5.30	4.70	4.00



(a) Hyper(A).

(b) Hyper(G).

(c) RBF(A).

(d) RBF(G).

Figure 3 – RMSE over time for varying reset strategies.

2.5.4 Comparison against other algorithms

In Table 5, we compare two variations of the ensemble techniques discussed in this paper against algorithms from the literature. Precisely, we use BAG$^{\text{f-l}}_{\mu}$-HTR$_p$ and BAG$^{a}_{\mu}$-HTR$_p$, which differ only on the reset strategy used. From these experiments we highlight that ARF-Reg tends to outperform all others in the synthetic datasets, including those with simulated concept drifts, while BAG$^{\text{f-l}}_{\mu}$-HTR$_p$ obtains the best results for the real datasets. We highlight that, contrary to what was observed in the experiments varying the reset strategy, the ARF-Reg algorithm, which includes an active drift detection strategy, was able to outperform other methods in the synthetic datasets that simulate concept drifts. However, if we compared it against the no-reset (n$_r$) from Table 4, it would not differ much in terms of RMSE. We also replicate the results for KNN in Table 5 to highlight how well it performs in comparison to algorithms specially designed to address evolving data streams, such as FIMT-DD, ORTO, and AMRules.

2.6 Conclusion

Ensembles are a popular approach in supervised learning since they improve performance by leveraging the predicting capabilities of a group of weak learners. Regression for evolving data streams, although relevant to many real-world applications and posing specific challenges, has not received as much attention by the research community as classification. In this paper, we study ensemble techniques for regression and show that, although performance is improved,

Table 5 – Comparing $\text{BAG}^{a}_{\mu}\text{-HTR}_{p}$ and $\text{BAG}^{\text{f-l}}_{\mu}\text{-HTR}_{p}$ against others.

Dataset	FIMT-DD	ORTO	AMRules	ARF-Reg	KNN	HTR_{m}	HTR_{p}	$\text{BAG}^{a}_{\mu}\text{-HTR}_{p}$	$\text{BAG}^{\text{f-l}}_{\mu}\text{-HTR}_{p}$
Abalone	2.6227	8.3230	2.3284	2.8277	2.3264	3.0540	2.8726	2.5719	**2.2506**
Bike	572.2625	2882.6485	134.5418	93.5983	**62.3067**	108.0877	85.1775	69.0090	68.3269
CalHousing	77589.2502	141419.9559	72436.8602	64253.7315	89876.9806	85204.2071	72327.5408	63307.2304	**62820.2290**
House8L	40945.7784	84042.0749	41388.1129	36325.3640	51046.4284	40883.3315	40874.6389	**35860.5305**	35966.3236
House16H	46798.5857	96237.2919	46072.4447	**39435.5565**	51164.6143	43890.7514	44301.9024	40028.5144	39461.4041
MetroTraffic	18719714.8607	6017208.9625	8798.4883	**1762.3839**	1945.0847	1951.2740	1910.3972	1858.1606	1842.0460
Pol	50.3320	90.6362	25.9851	**17.8487**	18.2383	26.4236	26.5224	36.1823	18.9284
Ailerons	0.0037	0.0070	0.0020	0.0002	**0.0000**	0.0000	0.0000	0.0000	0.0000
Elevators	0.3380	0.0715	0.0047	**0.0046**	0.0069	0.0054	0.0052	0.0048	0.0049
Fried	2.7390	7.8746	2.4735	**2.2410**	2.6746	2.7908	2.8103	2.9802	2.3569
MVDelve	2.9448	12.0426	3.8574	**1.5152**	8.3187	3.8745	3.8762	3.6996	2.1918
Hyper(A)	**1.8803**	15.8049	1.9713	3.3463	3.0400	5.9205	5.0287	4.5308	3.6148
Hyper(G)	**2.2675**	15.9225	2.3780	3.6790	3.3039	5.9094	5.0714	4.5717	3.7817
Hyper(I)	48.2369	126.0124	50.6482	**48.0818**	50.9431	79.8120	54.4025	65.4014	48.3118
RBF(A)	17.2946	57.5298	23.0847	**13.9592**	17.9073	23.2324	20.4353	14.8843	15.6089
RBF(G)	17.3575	58.0759	22.9520	14.9155	18.7157	23.1230	20.5220	**14.5837**	15.9212
RBF(I)	29.3239	38.9952	29.9269	28.3527	29.2988	**28.0289**	28.0496	28.4256	28.6819
Avg. rank	5.47	8.88	5.06	**2.59**	4.76	6.00	5.29	4.00	2.94
Avg. rank real	7.00	8.86	5.57	2.86	4.86	5.86	4.86	3.29	**1.86**
Avg. rank synth.	4.40	8.90	4.70	**2.40**	4.70	6.10	5.60	4.50	3.70

special considerations must be taken in the context of regression, e.g., combination techniques that integrate well with the base learner. To this end, we focused our analysis on techniques for training the base learners, combining predictions, the role of base learners, and the reset strategy that provides robustness against concept drifts. We conclude that resetting the base models has a positive effect in the predictive performance. Based on the experiments, we notice that a reactive strategy (based on a drift detector) may not produce the best results all the time. Simpler reset strategies such as periodically replacing members of the ensemble with new models trained on different windows can also boost performance in the ensemble. Another relevant observation was that random subspaces and random patches were not as effective for regression as when they were applied for classification.

For future works, we are considering a further analysis of ensembles of k-Nearest Neighbors for regression, and how to minimize the impact in the computational resources caused by an unbounded growth of the Hoeffding Tree algorithms.

CHAPTER

3

2CS: CORRELATION-GUIDED SPLIT CANDIDATE SELECTION IN HOEFFDING TREE REGRESSORS

Publication information: this chapter is an article published at Brazilian Conference on Intelligent Systems (BRACIS), whose proceedings are managed by Springer. According to the Springer Nature's Policy*, authors retain the right to reuse the content of a published paper in their own thesis.

Reference: MASTELINI, Saulo Martiello; PONCE DE LEON FERREIRA DE CARVALHO, André Carlos. 2CS: correlation-guided split candidate selection in Hoeffding tree regressors. In: **Brazilian Conference on Intelligent Systems.** Springer, Cham, 2020. p. 337-351.

3.1 Abstract

Incremental machine learning algorithms have been effective alternatives to deal with stream data. The Hoeffding Tree framework is one of the most successful solutions for supervised online prediction tasks. Although online regression tasks are present in several forms, and in many real-life problems, most of the research efforts have been devoted to classification. Existing regression tree solutions have strong limitations, mainly regarding their memory usage and running time. Hence, a new algorithm able to address these aspects in Hoeffding Tree Regressors is a relevant research issue. In this paper, we propose 2CS, a correlation-guided strategy to speed up Hoeffding Tree Regressor training. 2CS is conceptually simple and works by avoiding the exhaustive evaluation of all possible features as split candidates, as occurs in the existing solutions. Moreover, 2CS can be easily merged into existing incremental tree solutions and online

*<https://www.springer.com/gp/rights-permissions/obtaining-permissions/882>

tree ensembles algorithms, such as bagging and boosting. Throughout an extensive experimental evaluation, we show that the induction of 2CS-based models can be significantly faster than the traditional Hoeffding Tree Regressor algorithms, whereas retaining similar predictive power and memory use.

3.2 Introduction

Despite recent advances in technologies for data storage and processing, in many applications, such as big data, the amount of data being produced led to a situation where the computational power available to process all incoming data may not be enough. This occurs because most data is produced continuously, and fast, in the form of potentially unbounded streams (GAMA, 2010). Traditional supervised Machine Learning (ML) algorithms, i.e. *in batch* solutions, were not developed to operate in such circumstances (DOMINGOS; HULTEN, 2000; GAMA, 2010). Hence, more efficient solutions must be developed to cope with the requirements of big data (FAN; BIFET, 2013). These algorithms must process each incoming datum just once, and they cannot indefinitely store instances (GAMA, 2010; GOMES *et al.*, 2017).

In the last years, ML research in data streams has expanded at increasing steps. Among the supervised solutions for data stream mining, the Hoeffding Tree (HT) framework is one of the most explored (GAMA, 2010; IKONOMOVSKA; GAMA; DŽEROSKI, 2011b; GOMES *et al.*, 2017). Despite being primarily applied to classification tasks, there are adaptations of this framework for regression (IKONOMOVSKA; GAMA; DŽEROSKI, 2011a; OSOJNIK; PANOV; DŽEROSKI, 2018; MASTELINI *et al.*, 2019). However, dealing with continuous targets brings additional challenges for incremental algorithms. Differently from classification tasks, there is no well-defined target partitions, i.e., categories. Hence, tree solutions could potentially evaluate infinite data partition possibilities.

Practical HT solutions typically rely on the observed predictive features of the training instances to evaluate split decisions (IKONOMOVSKA; GAMA; DŽEROSKI, 2011b; IKONOMOVSKA; GAMA; DŽEROSKI, 2015). They store input feature values along with necessary statistics to guide these decisions. The required statistics can be incrementally maintained fairly efficiently (IKONOMOVSKA; GAMA; DŽEROSKI, 2011b; OSOJNIK; PANOV; DŽEROSKI, 2018). Nonetheless, the process of seeking for the best split candidates among the features of the problem is computationally expensive. The tree models have to test all features values stored between tree expansions as potential thresholds for new branches creation. The cost becomes even higher as the tree continues processing more instances and gathers more data. Hence, more efficient strategies for split candidate selection are needed for HT regressors (HTRs). More efficient solutions would benefit, for instance, ensemble algorithms, which have received increasing attention from the data stream mining community (IKONOMOVSKA; GAMA; DŽEROSKI, 2015; KRAWCZYK *et al.*, 2017; GOMES *et al.*, 2018). This is particularly true for

Boosting ensembles, as their additive nature make a parallel training difficult.

In this work, we investigate how to reduce the processing time needed to perform split decisions, without negatively impacting the prediction error and the required memory. For such, we use a simple yet effective heuristic to speed up split candidate selection in HTRs, named **C**orrelation-guided **S**plit **C**andidate **S**election (2CS). 2CS uses the correlation between numeric features and the target as a heuristic to rank predictive features. Hence, only a reduced subset of features is explored to expand the tree models, avoiding unnecessary computations. This strategy is conceptually simple and easily coupled within the HT framework. In this work, we are focused on stationary data streams, but in the future we intend to extend our analysis to non-stationary environments. Throughout an extensive experimental analysis, we show the capability of the 2CS-based trees of accelerating split decisions, while keeping predictive performance and memory use similar to the traditional HTRs.

The remaining of this text is organized as follows. In section 3.3 we present background information and important related work. In section 3.4 we formally introduce 2CS. In section 3.5 we describe the experimental setup used to compare the 2CS-based tree variants with the traditional HTRs. We show and discuss the obtained results in section 3.6. We make our final considerations and discuss possible directions for future research in section 3.7.

3.3 Background and Related Work

In this paper, we deal with data stream regression tasks, as defined next. We denote as S a possibly unbounded stream of instances in the form $S = \left\{(\mathbf{x}^t, y^t)\right\}_{t=1}^{\infty}$. Each instance $(\mathbf{x}^t, y^t)$ drawn at a timestamp t comes from an input space $\mathbf{X} \subset \mathscr{R}^m$, $m \in \mathbb{Z}^+$, and a target space $Y \subset \mathscr{R}$. The input space can also be referred to as feature space without loss of generality. Formally, a regression task can be formulated as the search for a function $f : \mathbf{X} \rightarrow Y$.

In data stream scenarios, we assume instances arriving continuously over time. Thus, f must be updated in an online fashion. Besides, in some cases we can expect an arbitrarily long delay before instances' labels are available for the incremental learning algorithms (GRZENDA; GOMES; BIFET, 2019). Nonetheless, in this study, we assume the true labels are available immediately after the model predicts the incoming instances.

Tree-based solutions have been extensively applied for data stream mining tasks, from which regression is not an exception (GAMA, 2010; IKONOMOVSKA; GAMA; DŽEROSKI, 2015; OSOJNIK; PANOV; DŽEROSKI, 2018). This comes from the fact that these models are conceptually simple and naturally interpretable (GAMA, 2010; BARDDAL; ENEMBRECK, 2019). As previously mentioned, the HT framework is the most prominent example of a tree-based algorithm family used in online prediction tasks. HTs rely on the Hoeffding Bound (HB) theorem (HULTEN; SPENCER; DOMINGOS, 2001; GAMA, 2010) to verify whether the model in training gathered enough evidence to enable its growth.

Nevertheless, research efforts on data stream mining were mostly devoted to classification tasks, whereas regression tasks were often overlooked (IKONOMOVSKA; GAMA; DŽEROSKI, 2011b; IKONOMOVSKA; GAMA; DŽEROSKI, 2015; GOMES *et al.*, 2018). Ikonomovska, Gama and Džeroski (2011b) proposed the Fast Incremental Model Tree with Drift Detection (FIMT-DD) algorithm, tackling for the first time regression tasks within the HT framework. In the same work, the authors also presented the Fast Incremental Regression Tree with Drift Detection (FIRT-DD). FIMT-DD uses linear perceptrons as leaf predictors, whereas FIRT-DD uses the target mean (IKONOMOVSKA; GAMA; DŽEROSKI, 2011b; IKONOMOVSKA; GAMA; DŽEROSKI, 2015). When coupled with the drift detection mechanism (indicated by the DD suffix), the tree algorithms use the Page-Hinkley (GAMA, 2010) test to detect concept drifts and grow alternate tree branches for the new concepts, very much alike a preceding HT solution for classification tasks (HULTEN; SPENCER; DOMINGOS, 2001). This algorithm is also similar to the Hoeffding Adaptive Tree (BIFET; GAVALDÀ, 2009), which also has a regression version (MONTIEL *et al.*, 2018). The HTRs were later on applied as base models for ensemble algorithms (IKONOMOVSKA; GAMA; DŽEROSKI, 2015; GOMES *et al.*, 2018) and adapted to multi-target regression tasks (IKONOMOVSKA; GAMA; DŽEROSKI, 2011a; OSOJNIK; PANOV; DŽEROSKI, 2018; MASTELINI *et al.*, 2019).

Both tree algorithms monitor the standard deviation reduction (SDR) in the target space as a measure for split recommendations (IKONOMOVSKA; GAMA; DŽEROSKI, 2011b; IKONOMOVSKA; GAMA; DŽEROSKI, 2015). When deciding whether and how to split, HTRs compare the SDR of the second-best split candidate divided by the SDR of the best one. HTRs verify whether this ratio plus the HB is smaller than 1 (IKONOMOVSKA; GAMA; DŽEROSKI, 2011b; MASTELINI *et al.*, 2019). In the affirmative case, we can state that the best split candidate is statistically better than the second one and, as a result, the tree creates new branches.

To decrease computational costs, HTs do not attempt to split after each incoming instance. Instead, they wait for n_{min} instances (also referred to as grace period) between split attempts. Further, in the case where split candidates are equally good, HTs also apply a tie-breaking mechanism to avoid indefinitely waiting for growth (GAMA, 2010; IKONOMOVSKA; GAMA; DŽEROSKI, 2011b). If the calculated HB shrinks below a tie-breaking threshold τ, a split is performed with the current best candidate.

In order to evaluate split candidates, HTRs rely on storing the observed feature values along with sufficient statistics related to them. The Extended Binary Search Tree (E-BST) algorithm is applied for this end (IKONOMOVSKA; GAMA; DŽEROSKI, 2011b). At each of its node, E-BST stores a set of sums related to the elements smaller (at the left side) and larger (at the right side) than each observed feature value. Hence, E-BST stores for the left and right sides the count of elements observed (n), the sum of the target values ($\sum y$), and the sum of squared target values ($\sum y^2$). These statistics are enough to calculate the variance and the standard deviation incrementally (IKONOMOVSKA; GAMA; DŽEROSKI, 2011b; IKONOMOVSKA;

GAMA; DŽEROSKI, 2015). Thus, each stored feature value can be evaluated as a potential split point.

When considering datasets with an increased number of features, evaluating all the observed split candidates becomes costly. HTRs calculate the SDR of each observed value for each feature. This action becomes even more impacting when considering ensembles of HTRs. In this work, we hypothesize that the features mostly correlated with the target should provide the best split points. Such idea was previously explored in batch scenarios (HOTHORN; HORNIK; ZEILEIS, 2006; SALEHI-MOGHADDAMI; YAZDI; POOSTCHI, 2011), but was not still covered in resource constrained online situations. In our proposal, the HTR traverse just the E-BST of the features most correlated with the target in search for split points. This action ought to decrease the processing time of HTRs without negatively affect their prediction power and memory consumption. This strategy, named 2CS, is detailed in the next section.

3.4 Correlation-guided split candidate selection

The split candidate selection in HTRs is guided by estimating the SDR of each partition candidate. In fact, the standard deviation is a measure of the spread of the data. Therefore, HTRs, similarly to traditional batch regression trees (BREIMAN *et al.*, 2017), aim at reducing the spread of instances lying in each of the created partitions. At the same time, regression trees try to make data partitions in such way that they become as maximally apart from each other as possible, following the "divide-and-conquer" principle (BREIMAN *et al.*, 2017; IKONOMOVSKA; GAMA; DŽEROSKI, 2011b).

In 2CS, we hypothesize that a measure of the relation between the numeric inputs and the target, such as the linear correlation, could give clues of which among them would be the most suited to perform a split decision (HOTHORN; HORNIK; ZEILEIS, 2006; SALEHI-MOGHADDAMI; YAZDI; POOSTCHI, 2011). This conjecture, to the best of our knowledge, was not explored yet in online learning scenarios. As presented in Gama (GAMA, 2010), we can easily calculate the linear correlation coefficient using a small set of incrementally maintained statistics. Most interestingly, almost all of them are inherently maintained by the HTRs. The correlation calculation is described in Equation 3.1.

$$r = \frac{\sum xy - \frac{(\sum x)(\sum y)}{n}}{\sqrt{\left(\sum x^2 - \frac{(\sum x)^2}{n}\right)\left(\sum y^2 - \frac{(\sum y)^2}{n}\right)}} \tag{3.1}$$

As previously mentioned in section 3.3, HTRs already maintain n, $\sum y$, and $\sum y^2$. Moreover, $\sum x$ and $\sum x^2$ are also maintained by HTRs to enable feature standardization (IKONOMOVSKA; GAMA; DŽEROSKI, 2011b; OSOJNIK; PANOV; DŽEROSKI, 2018; MASTELINI *et al.*, 2019). These measures refer to the sum of observed values for each feature and the sum of their squared

values, respectively. Here, for simplicity, we omit the indexing for each j-th feature. Therefore, the only measure missing from Equation 3.1 is $\sum xy$, i.e., the sum of the product between the feature values and the corresponding target observations. This additional measure can be easily added to the set of monitored statistics and just adds a constant increment in memory and time processing requirements. Now, we are ready to use the linear correlation as a heuristic to guide split candidate selection. It is important to mention that both positive and negative correlations are an indication of relationships between features and targets. Thus, in 2CS, we take the absolute value of the calculated correlations when ranking the features. Figure 4 summarizes 2CS's operation and how it fits within the HTR split decisions.

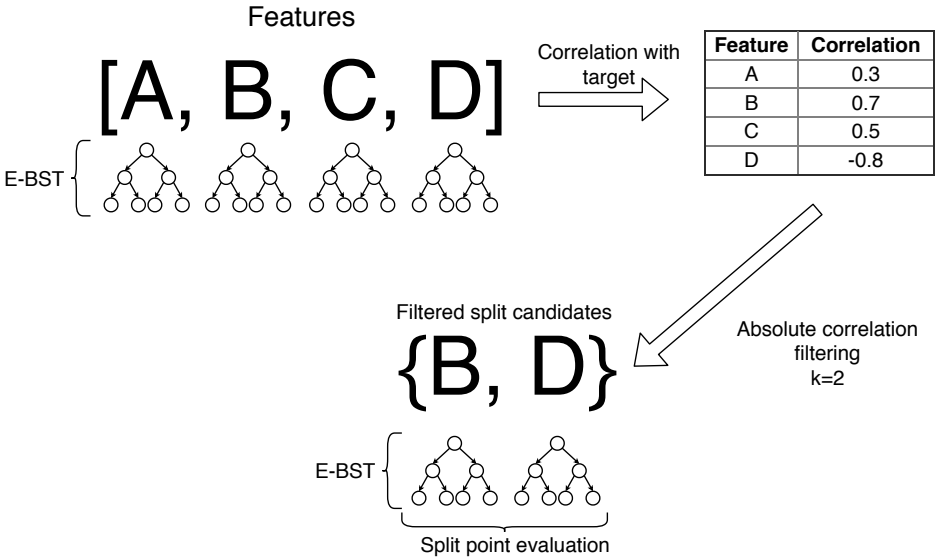


Figure 4 – Overview of 2CS's operation in the HTR framework.

The benefits of using a heuristic of how likely a feature will provide the best split decision are twofold. Firstly, different from batch algorithms, HTRs observe training examples incrementally as they arrive. Hence, at the moment of the splits, only partial information is available to make decisions. Although the HT framework gives us some guarantees that the trained models will perform similarly to batch ones, given enough observations (GAMA, 2010; IKONOMOVSKA; GAMA; DŽEROSKI, 2011b), shifting the tree growth too much towards what the currently available data describe can lead to overfitting (BISHOP, 2006; BARDDAL; ENEMBRECK, 2019). Secondly, by using a heuristic to select a subset of features to evaluate as split candidates, we can avoid performing possibly unnecessary processing efforts. The performance improvements are expected to increase jointly with the number of input features.

The complete functioning of 2CS is straightforward and easily included in the HTR framework. After $n_{\min}$ instances are observed by a leaf node, and the HTR is ready to attempt a split, our proposal performs the following three steps:

1. 2CS calculates the linear correlation between the numeric input features and the target using Equation 3.1;

2. The features are ranked according to the absolute value of their linear correlation;

3. 2CS selects the k most correlated features to evaluate as split candidates along with (possibly) existing nominal features.

Here, k is a hyper-parameter that must be adjusted. In our experimental evaluation, we compare different values for k and their impact on predictive performance (refer to section 3.5 for more information). It is important to note that the cost to calculate the correlations and rank the features accordingly to their values is usually negligible compared with the number of operations commonly performed in split attempts. This claim is supported by our experimental findings. Besides, measuring linear correlations only makes sense when both the features and the target are continuous. Hence, we only consider numerical features when applying the 2CS input feature filtering. Nominal inputs are treated following the strategy proposed by Osojnik, Panov and Džeroski (2018), where a tree branch is created for each category, in case the feature is used to split.

3.5 Experimental setup

In this section, we detail our setup to compare the traditional HTRs against the trees coupled with 2CS. They include the benchmark datasets, the settings for the tree-based algorithms, and the evaluation metrics. We performed the experiments using the *scikit-multiflow* Python framework for data stream mining (MONTIEL *et al.*, 2018). For such, we used a machine running a 64-bit Debian system with 128 GB of RAM and an Intel Xeon (X5690) CPU at 3.47 GHz.

3.5.1 Datasets

All the evaluated datasets were used in Ikonomovska, Gama and Džeroski (2011b), where FIMT-DD was first proposed. All of them are related to stationary tasks. The datasets vary from ≈ 4000 examples to ≈ 41000 instances, as shown in Table 6. The majority of the input features in these datasets are numeric. All the datasets are publicly available in platforms such as UCI[2] and OpenML[3].

3.5.2 Variants of 2CS

We evaluated different settings for 2CS, ranging the correlation rank threshold from $k = 2$ to $k = 5$. For instance, when $k = 2$, only the two numeric features most correlated with the target would be evaluated as split candidates. It is important to note, however, that the HTs

[2] <https://archive.ics.uci.edu/ml/datasets.php>
[3] <https://www.openml.org>

Table 6 – Datasets used in the experiments.

Dataset	#Examples	#Numeric Inputs	#Categorical Inputs
Abalone	4177	8	0
Ailerons*	13750	40	0
Elevators*	16599	18	0
House8L	22784	8	0
House16H	22784	16	0
MV*	40768	7	3
Pol	15000	48	0
Wind	6574	14	0
Wine	6497	11	0

* Synthetic dataset.

include a pre-pruning mechanism when performing splits. They also evaluate the possibility of not performing a split and maintaining the tree as it is, which here we refer as a *null* split option. Thus, HTRs with 2CS considered as split candidates the filtered numerical features, the null split, and the possibly existing categorical input values (as we discussed in section 3.4).

Here, we want to stress out that some of the evaluated datasets have fewer than 10 input features. This low number of features can reduce the processing time improvement brought by our proposal. We will return to this discussion in the result section.

3.5.3 Settings used in the tree predictors

During the experiments, we fixed some hyper-parameters for the tree algorithms with values commonly used in the literature (IKONOMOVSKA; GAMA; DŽEROSKI, 2011b; DUARTE; GAMA, 2015; OSOJNIK; PANOV; DŽEROSKI, 2018). Split attempts were performed at intervals of $n_{\min} = 200$ examples. We set the significance level of the HB calculation to $\delta = 10^{-7}$, and the tie-break hyper-parameter to $\tau = 0.05$.

Besides, in all cases, we used 200 examples to initiate the tree predictors, providing a "warm" start for the evaluations. Finally, the perceptron weights were started with uniform random values in the range $[-1, 1]$. In case of splits, new nodes inherit their ancestors' weights.

Regarding the decision tree induction algorithms, we considered regression versions of the HT framework, as available in *scikit-multiflow*. They operate very similarly to FIMT-DD/FIRT-DD but do not have concept drift adaptation mechanisms, as at this point we only deal with stationary streams. We denote by HTR_m and HTR_p tree models that use mean and linear perceptron as their prediction strategy, respectively. Although very similar, these tree algorithms have different prediction strategies and might react differently when working along with 2CS.

3.5.4 Evaluation strategy

In all the performed experiments, we used the *prequential* strategy for the evaluation of the HTR models (GAMA, 2010; KRAWCZYK *et al.*, 2017). In this benchmarking strategy, for each new incoming example, the model first makes a prediction and then learns from it. All tree-based algorithms were applied ten times to each dataset with different random number generator seeds. We report the average results obtained to reduce the effects of randomness and operational system external influences. For all the monitored metrics, we computed their mean value considering all the data seen until each measurement point and also considered windowed measurements. For such, we used a non-overlapping sliding window of size 200 (OSOJNIK; PANOV; DŽEROSKI, 2018).

We chose the Mean Absolute Error (MAE) as the error metric. This metric evaluates the absolute deviations of the tree's predictions compared to the expected target values. Furthermore, we report the amount of time spent by each algorithm (in seconds) and the total of memory resources consumed by the predictors (in MB).

We also performed statistical tests to verify whether the differences in the predictive performance of the models are statistically significant regarding the evaluation metrics. The Friedman test and the Nemenyi post-hoc test were applied with $\alpha = 0.05$, as described by Demšar (2006). We considered the windowed measurements for this end, to take into consideration the different time steps of the stream. We summed all the measurements for the different stream portions to obtain a single measurement per dataset.

3.6 Results and Discussion

In this section, we present and discuss our experimental results. First, we present the mean measurements after processing all the considered streams. Next, we discuss in details some interesting cases found during our analysis. Finally, we compare the performance of 2CS-based HTRs against traditional solutions, supported by statistical significance tests.

3.6.1 Overall results

We start presenting the MAE values for all the compared algorithm variants, in Table 7. The best variants obtained by the HTR algorithm are highlighted in **bold**. The best results per dataset are underlined. We indicate with $k = i$, $i \in \{2, ..., 5\}$, the variants of 2CS. The variant *all* represents the traditional HTR algorithms. The average ranks for the different algorithms are also indicated.

As shown in Table 7, in most cases, the 2CS variants performed very similarly to their traditional counterparts. Despite some ties, MAE differences were observed after the second decimal place, a piece of information here omitted for visual clarity. Nonetheless, in some cases,

such as the Pol dataset, the error difference was clear. These differences are a result of distinct tree structures generated by the 2CS strategy and the original HTR framework. In our proposal, correlation is applied as an additional heuristic to guide split selection and avoid excessive computations. Nevertheless, heuristics can sometimes be misleading, as some evaluated cases indicated. Anyhow, $\text{HTR}_p^{k=5}$ obtained the best overall ranking concerning MAE, being closely followed by HTR_m^{all}.

Table 7 – MAE results. The best results per algorithm are in **bold**, while the best results per dataset are underlined.

Dataset	HTR_m				
	all	$k=2$	$k=3$	$k=4$	$k=5$
Abalone	2.24 ± 0.00	$\mathbf{2.08 \pm 0.00}$	2.22 ± 0.00	2.40 ± 0.00	2.40 ± 0.00
Ailerons	$\underline{\mathbf{0.00 \pm 0.00}}$	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
Elevators	$\underline{\mathbf{0.00 \pm 0.00}}$	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
House8L	$\mathbf{21237.95 \pm 0.00}$	22466.70 ± 0.00	22010.95 ± 0.00	21965.50 ± 0.00	21933.58 ± 0.00
House16H	$\mathbf{24247.50 \pm 0.00}$	25984.76 ± 0.00	26058.76 ± 0.00	24711.80 ± 0.00	24877.13 ± 0.00
MV	$\mathbf{1.37 \pm 0.00}$	4.80 ± 0.00	4.42 ± 0.00	2.97 ± 0.00	2.35 ± 0.00
Pol	$\underline{\mathbf{12.62 \pm 0.00}}$	37.31 ± 0.00	37.31 ± 0.00	37.31 ± 0.00	37.31 ± 0.00
Wind	$\mathbf{3.63 \pm 0.00}$	4.30 ± 0.00	4.18 ± 0.00	3.71 ± 0.00	4.10 ± 0.00
Wine	$\mathbf{0.64 \pm 0.00}$	0.68 ± 0.00	0.68 ± 0.00	0.67 ± 0.00	0.65 ± 0.00
Average rank	3.33	7.67	7.78	6.44	6.67

Dataset	HTR_p				
	all	$k=2$	$k=3$	$k=4$	$k=5$
Abalone	1.66 ± 0.09	1.77 ± 0.12	1.67 ± 0.10	$\underline{\mathbf{1.58 \pm 0.07}}$	1.59 ± 0.08
Ailerons	0.00 ± 0.00	0.01 ± 0.00	0.01 ± 0.00	$\mathbf{0.00 \pm 0.00}$	0.00 ± 0.00
Elevators	0.02 ± 0.02	0.02 ± 0.01	0.01 ± 0.00	0.01 ± 0.01	$\mathbf{0.00 \pm 0.00}$
House8L	21411.20 ± 727.93	21403.59 ± 926.08	21144.45 ± 870.59	$\underline{\mathbf{20902.70 \pm 862.82}}$	20917.15 ± 856.20
House16H	24140.92 ± 1358.02	26152.65 ± 1559.79	26447.92 ± 1693.86	24315.09 ± 1368.92	$\underline{\mathbf{23987.59 \pm 1355.32}}$
MV	$\underline{\mathbf{1.18 \pm 0.22}}$	2.36 ± 0.38	2.12 ± 0.31	1.59 ± 0.26	1.64 ± 0.32
Pol	$\mathbf{14.25 \pm 1.38}$	27.88 ± 0.27	27.88 ± 0.27	27.88 ± 0.27	27.88 ± 0.27
Wind	3.44 ± 0.33	3.75 ± 0.46	3.45 ± 0.35	3.74 ± 0.45	$\underline{\mathbf{3.36 \pm 0.31}}$
Wine	0.62 ± 0.04	0.62 ± 0.04	0.62 ± 0.04	$\underline{\mathbf{0.61 \pm 0.03}}$	0.61 ± 0.03
Average rank	4.33	6.56	5.33	3.89	**3.00**

When considering the resulting model sizes, the 2CS-based variants generally originated models smaller than those generated by the original HTR algorithm. This fact is evidenced in Table 8. The best overall solution was $\text{HTR}_m^{k=4}$. Interestingly, the 2CS variants with smaller k hyper-parameter values did not necessarily result in less memory usage. This, again, comes from the fact that the correlation filtering for split candidates leads to different tree structures. In some cases, by evaluating fewer split candidates, the 2CS trees can take longer to split or split with increased frequency. We could not find a clear pattern relating the k value and resulting model size. Notwithstanding, excluding $\text{HTR}_p^{k=5}$, all the 2CS variants required, in general, less memory than the traditional HTR algorithms.

Table 9 presents the running times of the algorithms and their variants. This is the characteristic where the 2CS-based variants were clearly superior. Our proposal, as expected, was the fastest method. The improvements, nevertheless, were less pronounced for the datasets with less input features, as expected. When considering HTR_m, the running time increased with

Table 8 – Model size results (in MB). The best results per algorithm are in **bold**, while the best results per dataset are underlined.

Dataset	HTR_m					HTR_p				
	all	$k=2$	$k=3$	$k=4$	$k=5$	all	$k=2$	$k=3$	$k=4$	$k=5$
Abalone	1.71	2.79	2.24	0.68	**0.01**	1.71	2.80	2.25	0.69	**0.01**
Ailerons	12.09	**8.88**	10.45	9.88	13.01	12.10	**8.90**	10.47	9.90	13.03
Elevators	11.71	**1.10**	13.14	11.18	12.83	11.72	**1.10**	13.16	11.20	12.86
House8L	**28.30**	32.41	30.03	32.17	30.83	**28.33**	32.44	30.06	32.20	30.86
House16H	50.59	**48.91**	56.99	87.62	60.31	50.62	**48.95**	57.04	87.65	60.34
MV	61.74	77.12	66.99	**57.96**	71.38	61.80	77.17	67.04	**58.00**	71.42
Pol	9.89	**1.87**	1.87	1.87	1.87	9.94	**1.87**	1.87	1.87	1.87
Wind	8.39	10.93	7.82	8.00	**5.56**	8.40	10.94	7.82	8.01	**5.56**
Wine	5.53	4.37	4.25	**3.72**	4.65	5.53	4.38	4.26	**3.73**	4.66
Average rank	5.44	5.00	4.67	**3.89**	5.33	6.44	6.33	6.00	5.22	6.67

the increase of k, as highlighted by the average ranks. The prediction strategy of this tree variant is simple and does not require matrix operations, which reflected in the running time. On the other hand, the same did not occur for HTR_p. The fastest HTR_p variant was the the one with $k = 3$. Hence, we did not observe a clear relation between k and the final model runtime, as it occurs with HTR_m. The costs of updating the trees and making predictions ended up overcoming the gains in avoiding the evaluation of all features as split candidates.

3.6.2 Analysis

As previously mentioned, the use of correlation as a heuristic to guide split feature selection can lead to tree structures different from those induced by the original HTR algorithms. This different growth pattern can have either a positive or a negative impact on the predictive performance of resulting models. According to the experimental results, the use of 2CS usually improved the model size and running time. However, there are cases where 2CS uses more memory or even has a higher runtime than the original solution, e.g., when the 2CS-based trees are much larger than the original HTRs. At first glance, nonetheless, it seems improbable a tree structure built on a reduced amount of information results in lower prediction error. However, we observed this unusual behavior in our experiments. Next, we present our interpretation of this and other interesting cases. For such, we use two of the evaluated datasets.

First, we present the time-varying results for the Pol dataset, which has 48 input features. Due to space limits, we will focus on the HTR_m-based variants. As can be seen on the top chart of Figure 5a, the MAE values obtained by the 2CS variants were much worse than those obtained by HTR_m^{all}. The analysis of memory usage can give us evidence for this sub-par performance. As showed in the middle chart of the same figure, 2CS variants spent much less memory than the traditional HTRs variants. In fact, they presented a slowly increasing memory behavior. This steady memory increase is probably due to the 2CS-based models performing splits frequently,

Table 9 – Running time results. The best results per algorithm are in **bold**, while the best results per dataset are underlined.

Dataset	HTR_m				
	all	$k=2$	$k=3$	$k=4$	$k=5$
Abalone	2.39 ± 0.16	$\mathbf{1.14 \pm 0.02}$	2.07 ± 0.13	2.56 ± 0.02	4.74 ± 0.35
Ailerons	12.62 ± 0.43	$\mathbf{6.12 \pm 0.07}$	9.74 ± 0.29	6.19 ± 0.08	10.79 ± 0.33
Elevators	7.42 ± 0.45	$\mathbf{3.98 \pm 0.04}$	7.45 ± 0.23	4.97 ± 0.10	7.71 ± 0.33
House8L	13.98 ± 0.46	$\mathbf{7.25 \pm 0.12}$	11.34 ± 0.31	8.36 ± 0.17	12.90 ± 0.32
House16H	28.08 ± 0.71	$\mathbf{14.31 \pm 0.37}$	17.33 ± 0.42	21.48 ± 0.42	21.09 ± 0.54
MV	38.09 ± 0.54	$\mathbf{24.56 \pm 0.39}$	27.67 ± 0.46	30.73 ± 0.59	35.08 ± 0.52
Pol	8.00 ± 0.34	6.35 ± 0.18	6.40 ± 0.25	6.42 ± 0.29	$\mathbf{6.14 \pm 0.16}$
Wind	5.56 ± 0.24	$\mathbf{3.58 \pm 0.11}$	3.96 ± 0.21	3.91 ± 0.16	4.35 ± 0.16
Wine	$\mathbf{2.55 \pm 0.14}$	2.59 ± 0.11	2.76 ± 0.12	2.88 ± 0.20	2.96 ± 0.13
Average rank	4.67	**1.22**	2.89	3.11	4.56

Dataset	HTR_p				
	all	$k=2$	$k=3$	$k=4$	$k=5$
Abalone	3.49 ± 0.22	$\mathbf{2.71 \pm 0.10}$	3.03 ± 0.19	4.97 ± 0.23	5.69 ± 0.27
Ailerons	25.12 ± 1.08	22.44 ± 0.55	$\mathbf{22.01 \pm 0.57}$	22.08 ± 0.82	22.71 ± 0.52
Elevators	14.84 ± 0.55	$\mathbf{13.66 \pm 0.63}$	14.99 ± 0.52	15.11 ± 0.46	14.94 ± 0.30
House8L	19.62 ± 0.54	$\mathbf{16.61 \pm 0.22}$	17.05 ± 0.29	17.65 ± 0.31	18.61 ± 0.37
House16H	38.21 ± 0.79	$\mathbf{26.64 \pm 0.46}$	27.08 ± 0.66	30.79 ± 0.56	30.14 ± 0.54
MV	49.68 ± 1.18	$\mathbf{37.00 \pm 0.35}$	38.38 ± 0.37	42.19 ± 0.55	45.68 ± 0.43
Pol	28.59 ± 1.24	27.32 ± 0.81	$\mathbf{16.22 \pm 0.09}$	26.43 ± 0.84	16.26 ± 0.07
Wind	7.34 ± 0.63	5.94 ± 0.24	$\mathbf{3.81 \pm 0.06}$	5.99 ± 0.36	4.17 ± 0.02
Wine	4.15 ± 0.46	4.37 ± 0.40	$\mathbf{2.89 \pm 0.03}$	4.25 ± 0.62	3.04 ± 0.04
Average rank	9.11	6.89	6.00	8.56	8.00

so the E-BSTs did not gather much data. There are no memory drops, as they occur when a split is performed, and E-BSTs with multiple stored elements are discarded. In fact, E-BSTs are the main source of memory consumption, when compared to the other elements of the trees.

As expected, memory increase and drop occurs for HTR_m^{all}, as depicted in the figure. The bottom chart from Figure 5a shows that smaller models resulted in lower running times. In this case, we believe that 2CS misguided the tree growth, resulting in undesired performance levels. To overcome this deficiency, we intend to investigate more sophisticated mechanisms for split candidate selection. A potential strategy to pursue would be applying meta-learning for split candidate recommendation (BRAZDIL *et al.*, 2008).

The second case discussed refers to the Elevators dataset, whose performance profiles are presented in Figure 5b. This time, we will only consider the HTR_p variants. Regarding MAE, after ≈ 8000 instances, the 2CS-based HTR_p become more accurate than HTR_p^{all}. Interestingly, the higher the k, the faster the 2CS-based variants reacted to the sudden error increase. Even the most limited $\text{HTR}_p^{k=2}$ was better than HTR_p^{all}. In this context, we observed the opposite of what we saw in the previous case. Our proposal avoided overfitting to the current observed state, as it selected better features for the splits. HTR_p^{all} relies on the currently stored values in the

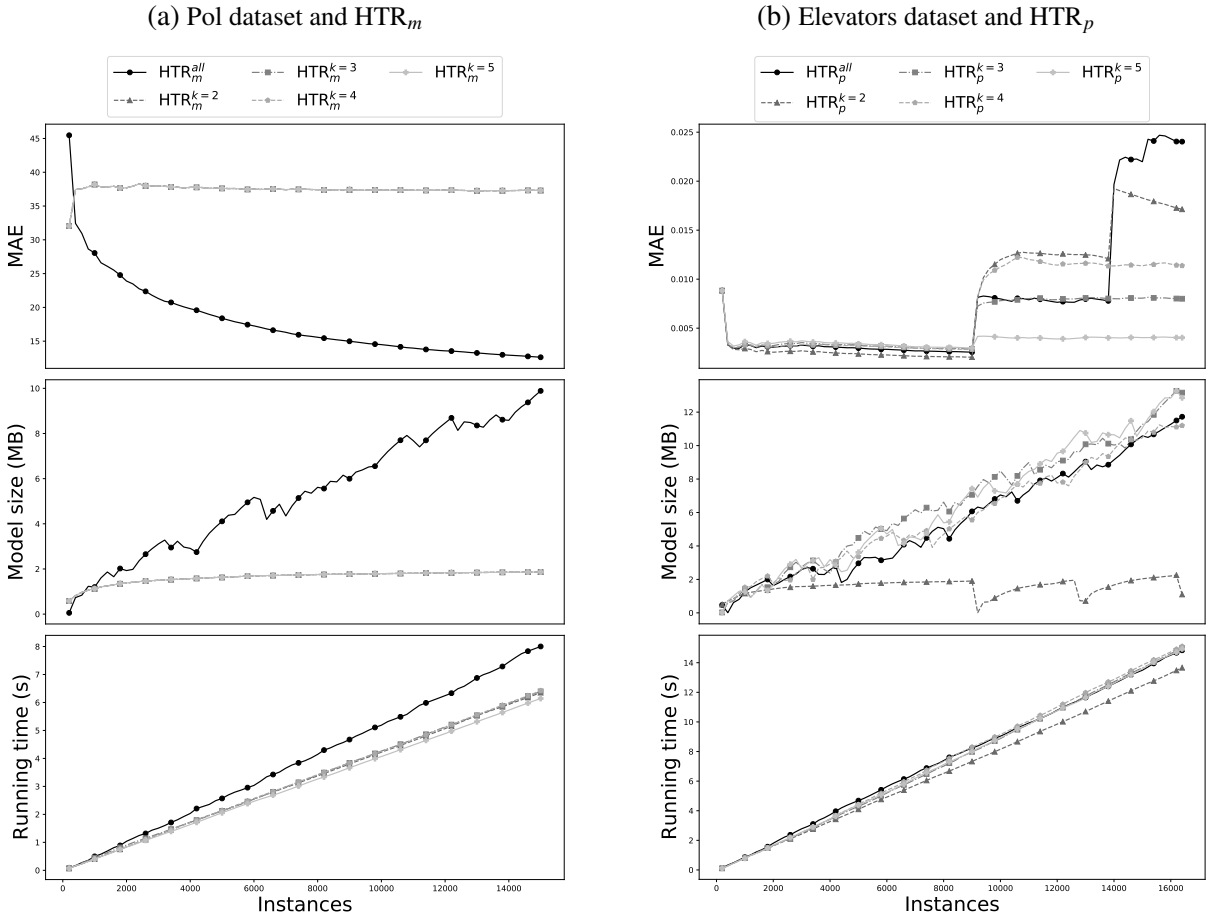


Figure 5 – Time varying results for the Pol and Elevators datasets.

E-BST to decide the splits. However, the current best split decision might be only valid for the data observed so far. The best split for future data might be different from the current estimation. This seems to be the case for the Elevators dataset. Nevertheless, our proposal's variations were able to overcome this problem. This type of situation deserves future investigation. We intend to put special attention to non-stationary problems to evaluate how 2CS might affect the tree algorithms in these scenarios.

Regarding memory and processing time, the 2CS-based variants performed very similarly to HTR$_m$ in the Elevators dataset. 2CS generated faster trees than HTR$_p^{all}$. They also used varying, but similar memory amounts to HTR$_p^{all}$. The only clear exception was the memory usage of HTR$_p^{k=2}$. Following the same reasoning used for the Pol dataset, the 2CS variant splits with increased frequency in comparison with the other tree models, which resulted in reduced memory usage. Nonetheless, this time, HTR$_p^{k=2}$ obtained smaller errors than HTR$_p^{all}$.

3.6.3 Statistical analysis and 2CS variant selection

We present our statistical analysis using critical difference (CD) diagrams (Figure 6). Regarding MAE, we did not observe a clear pattern for both HTR$_m$ (Figure 6a) and HTR$_p$ (Figure 6b). The 2CS trees performed comparably to their original counterparts, with a few

exceptions. In general, the higher the k, the smaller the error. The memory usage was statistically equivalent among all the compared algorithms. When comparing the runtime of the models, however, we observed that some 2CS variants (usually the ones with $k \leq 3$) were significantly faster than their vanilla versions. In the future we intend to increase the number of datasets to obtain more pieces of evidence for comparison and highlight the differences between the algorithms.

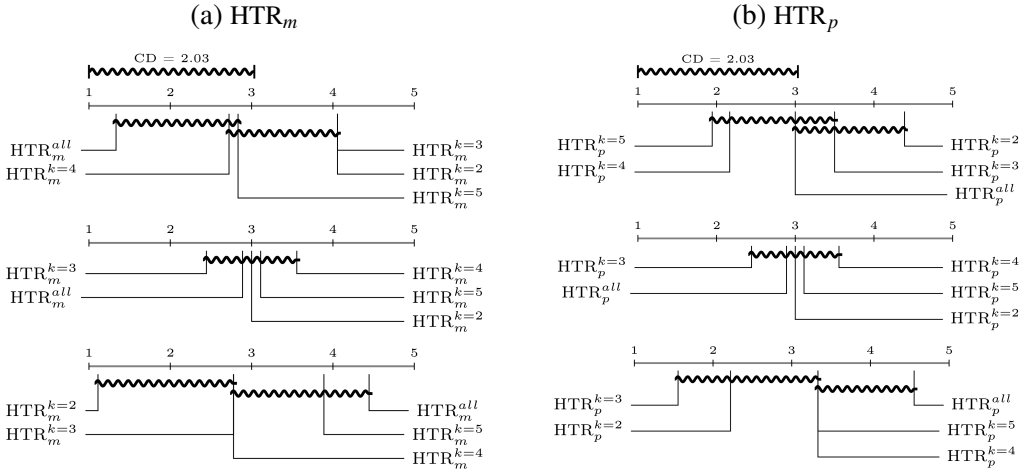


Figure 6 – Statistical tests results: MAE (top), Model size (middle), Running time (bottom). Tree algorithms whose ranks do not differ by at least the critical distance (CD) value are considered statistically equivalent (at $\alpha = 0.05$).

With many algorithm variants and evaluation metrics, it might be difficult to select models that present a good compromise between error, memory usage, and running time. We generated a Principal Component Analysis (PCA) biplot (GABRIEL, 1971) to comprise all algorithms and metrics under evaluation on the same chart, as presented in Figure 7. In the figure, points represent the compared algorithms, and vectors represent the normalized evaluation metrics. The direction of the metrics indicates their influence over the algorithms, i.e., the farther the points are from the origin, the highest their MAE, Memory usage, or Running time (depending on their placement in relation to the metrics' vectors). Lastly, the angle between the metrics is an indication of correlation, in case the vectors have roughly the same or opposite directions (which configure positive and negative correlations, respectively). Orthogonality is an indication of no correlation.

The obtained biplot enables us to draw interesting observations. Firstly, MAE and the Running time are negatively correlated, i.e., the higher the MAE, the smaller the time spent by the trees, and vice-versa. The resulting tree size does not seem to be related to the error nor the running time of the models. The least accurate variant was HTR$_m^{k=2}$, the slowest ones were HTR$_p^{k\in\{4,5,all\}}$, and the biggest ones were the vanilla HTR versions. Towards the origin of the chart we have variants that offer the best compromise between the tree metrics: HTR$_m^{k=5}$ and HTR$_p^{k\in\{2,3\}}$. We could choose one among them based on which metric is the priority in a specific application.

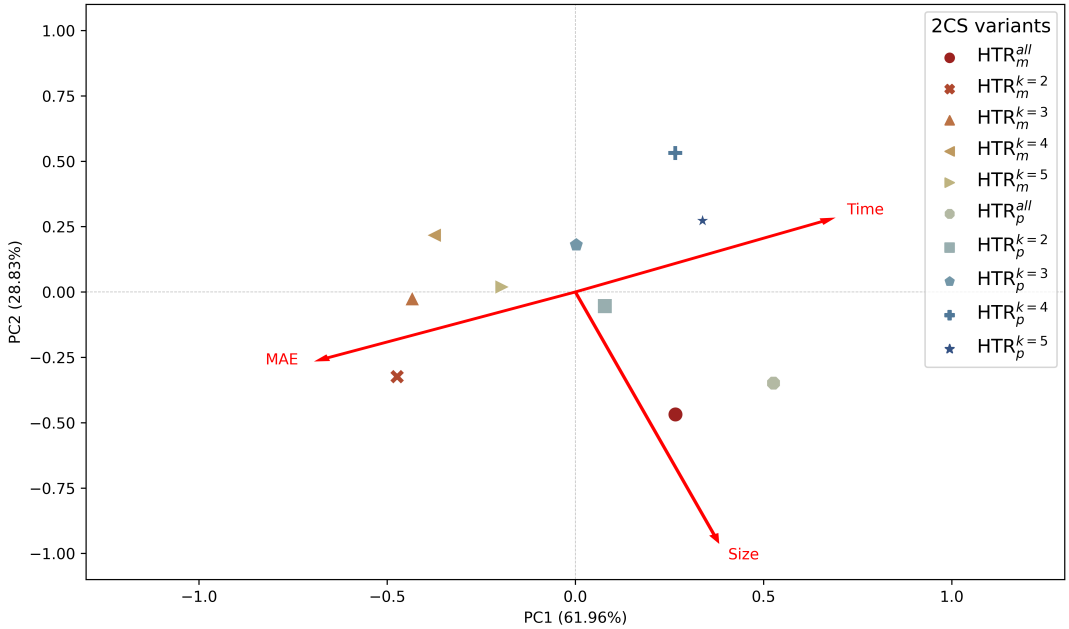


Figure 7 – PCA biplot comprising all the compared algorithms and evaluation metrics.

3.7 Final considerations

Hoeffding Trees represent one of the most prominent algorithmic solutions for incremental supervised learning. They are conceptually simple, easy to interpret, and usually fast to train. Online regression brings additional challenges as there are no well-defined or trivial target partitions to guide tree growth, as in classification. Hence, for regression streaming applications, more efficient strategies for feature split point selection are needed.

In this work, we proposed the use of linear correlation as a complementary heuristic to select a subset of input features to evaluate as split candidates. Our proposal, 2CS, can be easily merged into the HTR framework, has just a single hyper-parameter to tune, and does not increase the amount of the necessary memory. 2CS reduces the processing time by evaluating fewer input features as split candidates. Experimental results showed that 2CS retain the prediction capabilities of vanilla HTRs, whereas using a similar memory footprint and being significantly faster.

As future work, we intend to evaluate the 2CS-based trees as base models for ensemble algorithms and to consider non-stationary problems. We also intend to evaluate the capabilities of 2CS by using larger regression datasets regarding both their number of observations and input features. We also plan to apply more sophisticated strategies for split candidate selection. One of the possible techniques to explore is the usage of meta-learning to recommend the best split candidate.

Acknowledgements

The authors would like to thank FAPESP (São Paulo Research Foundation) for its financial support (grants #2018/07319-6, #2016/18615-0 and #2013/07375-0) and Intel Inc. for providing equipment for some of the experiments.

CHAPTER

4

USING DYNAMICAL QUANTIZATION TO PERFORM SPLIT ATTEMPTS IN ONLINE TREE REGRESSORS

Publication information: This chapter is an article published in the Pattern Recognition Letters journal, by Elsevier. According to the publisher's policies, we retain the right to include it in this thesis for non-commercial purposes*.

Reference: MASTELINI, Saulo Martiello et al. Using dynamical quantization to perform split attempts in online tree regressors. **Pattern Recognition Letters**, v. 145, p. 37-42, 2021.

4.1 Abstract

A central aspect of online decision trees is evaluating the incoming data and performing model growth. For such, trees much deal with different kinds of input features. Numerical features are no exception, and they pose additional challenges compared to other kinds of features, as there is no trivial strategy to choose the best point to make a split decision. Regression tasks are even more challenging because both the features and the target are continuous. Typical online solutions evaluate and store all the points monitored between split attempts, which goes against the constraints posed in real-time applications. In this paper, we introduce the Quantization Observer (QO), a simple yet effective hashing-based algorithm to monitor and evaluate split candidates in numerical features for online tree regressors. QO can be easily integrated into incremental decision trees, such as Hoeffding Trees, and it has a monitoring cost of $O(1)$ per instance and a sub-linear cost to evaluate split candidates. Previous solutions had a $O(\log n)$ cost per insertion (in the best case) and a linear cost to evaluate split candidates. Our extensive experimental setup highlights QO's effectiveness in providing accurate split point suggestions

*<https://www.elsevier.com/about/policies/copyright>

while spending much less memory and processing time than its competitors.

4.2 Introduction and Background

With the ever growing production of data, data stream mining became a particularly relevant research area. Data streams might come from different sources, such as the internet, Internet of Things (IoT) devices, sensors, among others. They are potentially unbounded and might change through time, thus, multiple specialized unsupervised and supervised learning algorithms have been proposed to tackle data streams. Regarding supervised data stream learning, online or incremental Decision Tree (DT) algorithms and ensembles thereof are frequent option among researchers and practitioners (KRAWCZYK *et al.*, 2017; BIFET *et al.*, 2018). DTs are flexible, interpretable, and do not make any assumptions about the data's characteristics.

However, online DT (ODT) models face additional constraints when compared with their traditional batch counterparts. First, while the data is unbounded, the computational resources are limited. For this reason, ODTs can neither store instances indefinitely nor process them multiple times (GAMA, 2010; BIFET *et al.*, 2018). Typical solutions process each incoming datum once, which is then discarded. Besides, from the start, the trees must be able to predict new instances and be updated anytime. Hence, ODT models should be maximally accurate, whereas keeping the memory and processing time usage minimal.

Multiple families of theoretical ODT algorithms have been proposed over the years. The most popular of them, the Hoeffding Tree (HT), relies on Hoeffding's inequality (HOEFFDING, 1963) to decide when an incremental model has gathered enough information to expand itself. Instances of HTs were proposed for classification (DOMINGOS; HULTEN, 2000), regression (IKONOMOVSKA; GAMA; DŽEROSKI, 2011b), structured output tasks (OSOJNIK; PANOV; DŽEROSKI, 2018). Other ODTs variants also use the same theoretical framework (PFAHRINGER; HOLMES; KIRKBY, 2007), as well as decision rule systems (ALMEIDA; FERREIRA; GAMA, 2013). ODT solutions that do not fit in the HT framework were also explored (RUTKOWSKI *et al.*, 2012; GOUK; PFAHRINGER; FRANK, 2019).

Apart from their core differences, ODTs share a common property: they monitor input features and perform split attempts. ODTs must process the stream's features and store statistics relating each input to the target value as data continuously arrives. These statistics differ for classification or regression tasks (DOMINGOS; HULTEN, 2000; IKONOMOVSKA; GAMA; DŽEROSKI, 2011b). The stored statistics enable the models to evaluate split candidates and decide upon the best feature (and split/cut point) to expand their structure. For such, ODTs rely on a class of algorithms named Attribute Observers (AO), and carry one AO per feature in each one of their leaves.

ODTs can efficiently deal with nominal attributes since split enabling statistics can be directly maintained for each category. Numerical attributes, on the other hand, do not have

explicit partitions and cannot be trivially manipulated to calculate split points (PFAHRINGER; HOLMES; KIRKBY, 2008; IKONOMOVSKA; GAMA; DŽEROSKI, 2011b). Usually, batch DT rely on sorting operations to evaluate split candidates. At each node, the tree has to sort the numerical input values and evaluate every available binary split decision.

In the online setting, the cost of performing sorting operations is prohibitive. Thus, less computationally expensive alternatives have been proposed to overcome this limitation (PFAHRINGER; HOLMES; KIRKBY, 2008; GOUK; PFAHRINGER; FRANK, 2019). Typical ODT solutions use data structures or attribute distribution estimators to keep the input values and their statistics sorted with reduced costs. Early ODTs for classification and most of the current versions for regression use a binary search tree (BST) as AO. This structure is named Extended Binary Search Tree (E-BST) since it stores both input values and target statistics in its nodes. While classification E-BSTs store class counts in their nodes (DOMINGOS; HULTEN, 2000), regression E-BSTs keep variables used for online mean and variance calculation (IKONOMOVSKA; GAMA; DŽEROSKI, 2011b; OSOJNIK; PANOV; DŽEROSKI, 2018).

Although fairly efficient, E-BST still has an insertion cost of $O(\log n)$ per observation, in the best case, and a memory cost of $O(n)$, where n is the number of stored instances. An in-order traversal of the BST is needed to account for all monitored input values and evaluate split candidates. Such traversal, or split point query, has a cost of $O(n)$. Moreover, only partial information is available for evaluation at the split time, in contrast to batch DT solutions. Consequently, the split chosen for an input attribute after observing n instances might not be the same had the tree monitored $n+k$ instances. If extrapolation can be applied, the obtained split points might be improved.

More efficient numerical AOs have been investigated for classification (PFAHRINGER; HOLMES; KIRKBY, 2008). In these tasks, the target attribute has well-defined partitions (categories) and more effective strategies can be explored. Histograms are used as AOs (PFAHRINGER; HOLMES; KIRKBY, 2008), and bear a cost of $O(\log m)$ per insertion and $O(m)$ of memory and query costs, where m represents the number of histogram bins. Another popular AO approximates the probability density distribution of each class using Gaussian distributions (PFAHRINGER; HOLMES; KIRKBY, 2008). These distributions can be easily constructed in an online fashion and only require estimating the sample mean and variance. This strategy has a $O(1)$ cost per insertion, and a sub-linear query cost.

Unfortunately, in regression tasks, more than counts are required to calculate the dispersion measure for split candidate evaluation. The Variance Reduction (VR) strategy applied to evaluate how promising is each split point involves keeping incremental estimations of the sample mean and variance. These estimators must be kept for each split candidate lying in the hyper-rectangle defined by a path from the tree root to a leaf. Therefore, incremental tree regressors must be able to calculate variances for each partition created by dividing the input space in axis-aligned splits of the form $x \leq c$ (left branch) and $x > c$ (right branch), where x is

one of the input features, and c is the split threshold (or cut value).

Most of the existing regression ODTs still rely on the E-BST structure, whose disadvantages were previously discussed. Besides, the algorithm commonly used in the E-BST (IKONOMOVSKA; GAMA; DŽEROSKI, 2011b) for calculating variances is known to be unstable and produce inaccurate results (FINCH, 2009; KNUTH, 2014). Consequently, all the current regressors relying on E-BST are prone to intensive memory usage and processing time, as well as yielding inaccurate results due to the numerical instability of the incremental variance estimators. Ideally, we would like to obtain AO algorithms for regression whose cost for adding new examples and querying split candidates is $O(1)$. However, if we can devise an accurate solution with constant insertion cost and sub-linear query costs, this is already on par with the most advanced AO solutions for ODT classifiers.

This paper introduces the Quantizer Observer (QO), a dynamical quantization algorithm to handle numerical features in ODT regressors. QO stores the same kind of targets' statistics as those monitored by E-BST. However, QO has a $O(1)$ cost per insertion of elements and a memory cost of $O(n')$, where $n' \ll n$. Despite being much faster than E-BST, QO can still produce split candidates with similar discriminating capability. To assess its performance, QO is experimentally compared with the traditional E-BST and a variation of E-BST that truncates the input values before their insertion in the BST, named Truncated E-BST (TE-BST). TE-BST aims to reduce the memory usage and processing time of E-BST.

To circumvent the problem of inaccurate incremental variance estimation, we extended the formulae proposed by Chan, Golub and LeVeque (1982) to handle robust and distributed variance estimation. The expressions proposed by Chan, Golub and LeVeque (1982) enable summing partial estimates of the variance. We extend them by also enabling subtracting partial estimates of variance from each other. All the AOs for regression compared in this work adopt these enhanced and robust incremental variance estimators.

We benchmark the different AOs using an extensive synthetic data setup and account for insertion, storage, and query costs. Here, we focus on the AOs rather than on the actual tree models. This focus allows us to isolate the splitting procedure from other aspects of the ODTs, such as tree traversal and how the models compute predictions. We vary the sample characteristics, size, and noise levels to simulate different situations the AOs might face when working in the trees. According to the experimental results, QO reduced, with statistical significance, the memory costs and processing time when compared with the existing AOs for regression. The experimental results also show that QO can produce split points similar to those provided by E-BST.

The remaining of this work is organized as follows: section 4.3 formalizes the split point search in regression ODTs. Section 4.4 presents the robust incremental mean and variance estimators that replace the previous used unstable estimators. Next, section 4.5 presents our proposed AO, QO. We detail our evaluation setup in section 4.6 and discuss the obtained results

in section 4.7. Finally, we present our final considerations and possible directions for future research in section 4.8.

4.3 Problem definition

Suppose an infinite stream $S = \{(\mathbf{x}, y)_t\}_{t=0}^{\infty}$, where each object at time t, $(\mathbf{x}, y)$, is composed of numerical input attributes $x \in \mathbf{x}$, and a scalar target attribute y. Regression ODT induction algorithms work by creating binary partitions in the numerical features. These partitions take place at a specific attribute value $x = c$. Hence, trees grow by creating branches from a decision node d to leaf nodes l_- and l_+, which are defined by the tests $x \leq c$ and $x > c$, respectively. To guarantee that the resulting models will be accurate, the tree learning algorithms have to determine the best (x, c) combination, among all $x \in \mathbf{x}$.

As previously discussed, DT regressors typically minimize the MSE of the target value of points in a leaf node compared to their mean value, i.e., the centroid or prototype point. This strategy is equivalent to minimizing the variance of the y values belonging to each leaf. For this reason, when performing a split attempt, DT regressors aim at choosing the partition candidate that maximally reduces the variance of y, here simply referred to as s^2.

The resulting heuristic to guide tree growth, called Variance Reduction (VR), is defined in Equation 4.1.

$$\mathrm{VR}_{(d,\{l_-,l_+\})} = s^2(d) - \frac{|l_-|}{|d|} s^2(l_-) - \frac{|l_+|}{|d|} s^2(l_+) \tag{4.1}$$

The notation $|.|$ represents the number of instances lying in the tree node inside the brackets. From Equation 4.1, it is clear that trees must be able to calculate the variance of the target variable in their nodes. This operation is trivial in batch regression DT induction algorithms since all data is available beforehand. In online applications, however, the algorithms must estimate the variance incrementally and at any time for each partition induced by a given realization of (x, c). AOs, such as E-BST, are used for this end.

Each node in an E-BST represents one of the observed values x_v of the monitored feature x. New observations are added as new leaves in the E-BST, and the order in which the x values are inserted impact how balanced the BST becomes. Only the nodes accessed when a new instance is sorted down the E-BST have their statistics updated. Nodes store target statistics accounting for all x observations that are smaller than or equal to their x_v.

Originally, E-BST was designed to operate using the so-called naive incremental variance estimator (KNUTH, 2014). Hence, each E-BST node stores $\sum_{x \leq x_v} w$, $\sum_{x \leq x_v} y$, and $\sum_{x \leq x_v} y^2$, respectively, the sum of weights, the target values, and the squared target values. These properties, although producing inaccurate estimates of the variance and being prone to numerical cancellation (FINCH, 2009), can be easily merged. In other words, we can either add or subtract

the statistics of two different E-BST nodes and calculate the resulting variance. Finally, by doing a complete in-order traversal, we can retrieve the statistics necessary to compute the VR value for the partition induced by each x_v in the E-BST.

Next, we discuss how to improve the VR values calculated in the E-BST and any other AO for regression tasks.

4.4 Robust Variance calculation

Calculating the variance of each candidate partition is a central aspect of ODT regressors. As previously mentioned, the current solutions rely on an incremental algorithm with well-known problems (FINCH, 2009; KNUTH, 2014). We start this section by describing the Welford's algorithm, a robust and popular alternative for calculating variance incrementally (KNUTH, 2014).

The Welford's algorithm works by keeping an estimate of the sample mean at the n-th instance, $\bar{x}_n$, which is used to update the auxiliary second order statistics $M_{2,n}$. This auxiliary value is used, in turn, to calculate the variance. At the beginning of the data monitoring process, we set both $\bar{x}_0$ and $M_{2,0}$ to zero. After each new observation x_n arrives, we update the stored statistics, as follows. The mean estimate update is given by Equation 4.2.

$$\bar{x}_n = \bar{x}_{n-1} + \frac{x_n - \bar{x}_{n-1}}{n} \tag{4.2}$$

The $M_{2,n}$ value is also recursively updated by using Equation 4.3.

$$M_{2,n} = M_{2,n-1} + (x_n - \bar{x}_{n-1})(x_n - \bar{x}_n) \tag{4.3}$$

At any time, one can get an estimate of the monitored sample variance by calculating $s_n^2 = \frac{M_{2,n}}{n-1}$, for $n > 1$.

Chan, Golub and LeVeque (1982) extended the presented formulae to handle parallel updates. In other words, by using the new expressions presented next, we can process different parts of the stream separately and then merge the resulting statistics to obtain mean and variance estimates for the whole sample.

In the following equations, for notation simplicity, we do not show the n subscript. Instead, we add the subscripts A and B to denote two groups of partially monitored data, whose statistics are going to be merged. We also denote by AB the resulting merged group. The total number of observed examples can be directly computed as $n_{AB} = n_A + n_B$. By using n_{AB}, we can calculate the total estimate of the mean, using Equation 4.4.

$$\bar{x}_{AB} = \frac{n_A \bar{x}_A + n_B \bar{x}_B}{n_{AB}} \tag{4.4}$$

Now, we have all the needed tools to estimate the total second order statistic, as defined in Equation 4.4. In the expression, $\delta = \bar{x}_B - \bar{x}_A$.

$$M_{2,AB} = M_{2,A} + M_{2,B} + \delta^2 \frac{n_A n_B}{n_{AB}} \tag{4.5}$$

By doing some simple algebraic manipulations in the equations of Chan, Golub and LeVeque (1982), we obtain expressions to govern the subtraction of the incremental statistics. In other words, one can get the complement of partially monitored sample statistics if they also have complete statistics. Therefore, we get the same addition and subtraction properties the naive incremental variance calculation algorithm has, but still retain superior accuracy in our estimates.

We start by the simplest one, the number of objects, which is given by $n_A = n_{AB} - n_B$. Equation 4.6 presents the expression to retrieve $\bar{x}_A$, given $\bar{x}_{AB}$ and $\bar{x}_B$.

$$\bar{x}_A = \frac{n_{AB}\bar{x}_{AB} - n_B\bar{x}_B}{n_A} \tag{4.6}$$

Finally, we can get the complement of partial second order statistic by using Equation 4.7.

$$M_{2,A} = M_{2,AB} - M_{2,B} - \delta^2 \frac{n_A n_B}{n_{AB}} \tag{4.7}$$

4.5 Quantizer Observer

Our proposal, QO, is inspired by Locality Sensitive Hashing (LSH) (DATAR *et al.*, 2004) algorithms, which are used to approximate nearest neighbor search and also discretize numerical input features. Our proposal also aims at creating partitions so that similar input values are grouped. Unlike most LSH algorithms, QO relies on a single hash structure, H, to create hash slots (also referred to as buckets) for the discretized features. Moreover, QO deals with one feature at a time, so there is no need to involve multiple random projections to define hash codes, as in popular LSH solutions. Instead, we simply define a quantization radius, r, to discretize the incoming input feature. In each of H's slots, QO keeps the sum of x's values, and estimations of the mean and variance of y. QO relies on the equations presented in section 4.4 to update and combine the target's statistics.

The functioning of QO is straightforward and can be easily incorporated into existing regression tree ODT algorithms. QO works as follows. For each i-th observation of a feature x, we select its corresponding hash code h, i.e., the slot it belongs to in H, by following the simple projection scheme $h = \left\lfloor \frac{x_i}{r} \right\rfloor$. If h is not in H, we create a new slot to accommodate the incoming data, otherwise, we include the values of x_i and y_i to the existing slot. Algorithm 1 illustrates the update procedure of QO, i.e., how AO monitors incoming examples.

Algorithm 1 – QO update.

Input:
r: the quantization radius.
S_x: stream with one numerical feature x and the target y.
Initialization:
Let H be an empty hash table.

for $x_i, y_i \in S_x$ **do**
  Let $s^2_{y_i}$ be a variance estimator with one observation, y_i.
  Let $h \leftarrow \lfloor \frac{x}{r} \rfloor$ be chosen hash slot.
  if $h \in H$ **then** ▷ Update statistics and prototype
    $H[h]_x \leftarrow H[h]_x + x_i; \quad H[h]_{s^2} \leftarrow H[h]_{s^2} + s^2_{y_i}$
  else ▷ Create a new hash slot
    $H[h]_x \leftarrow x_i; \quad H[h]_{s^2} \leftarrow s^2_{y_i}$
  end if
end for

Since h is directly proportional to x_i, when evaluating split candidates, QO sorts the keys stored in the hash to get an ordered representation of the whole sample. We retrieve the necessary information to calculate the VR statistic by computing the cumulative sum of the ordered H's elements. Hence, the split candidate query cost is $O(n' \log n')$, where n' is the number of slots in H. We experimentally observed that $n' \ll n$, where n is the total number of observations.

Split points are defined as the average between the prototype attribute values of two consecutive slots in the ordered hash. We define the prototype as the mean of the x values belonging to a slot. Other strategies could also be employed, such as interpolating consecutive slots with a regression model. Nonetheless, to reduce computational costs, we opted for using a simple approach. The prototype feature value can be easily obtained using the sum of x's values and the number of observations in each slot. We illustrate the split point query of QO in Algorithm 2.

4.6 Experimental setup

This section describes the simulation protocol used in this study to compare QO against E-BST and TE-BST, as well as the evaluation metrics and settings used in the AOs.

4.6.1 Simulation protocol

We evaluated the effectiveness of the AOs when monitoring data samples of varying sizes, whose targets were defined by different functions. After calculating the target's values, the inputs were also (in some cases) subject to different levels of noise. The AOs processed the data sequentially, one instance at a time. After processing the whole sample, the AOs calculated the best split candidate they could provide, given their inner structures.

Algorithm 2 – QO split candidate query.

Input:
H: an existing QO realization of feature x.
s_y^2: the variance estimation of the target y.
Return:
c: the best found split threshold in x.
c_{vr}: the VR value obtained by partitioning x at c.

Let s_{aux}^2, a variance estimator initialized with zero.
$c_{vr} \leftarrow \varnothing$; $x_{\text{aux}} \leftarrow 0$; $i \leftarrow 0$
for h in `sorted`(H) **do**
  if $i > 0$ **then**
    Let $x_p \leftarrow \dfrac{H[h]_x}{H[h]_n}$ be the current prototype x value.
    Let $\hat{c} \leftarrow \dfrac{x_{\text{aux}} + x_p}{2}$ be the candidate split point.
    Let $\hat{c}_{vr}$ be the VR obtained from $\left\{s_y^2, s_{\text{aux}}^2, s_y^2 - s_{\text{aux}}^2\right\}$.
    if $c_{vr} = \varnothing$ **or** $\hat{c}_{vr} > c_{vr}$ **then**
      $c_{vr} \leftarrow \hat{c}_{vr}$; $c \leftarrow \hat{c}$
    end if
  end if
  $x_{\text{aux}} \leftarrow x_p$; $s_{\text{aux}}^2 \leftarrow s_{\text{aux}}^2 + H[h]_{s^2}$; $i \leftarrow i + 1$
end for

Table 10 summarizes the settings used in our data generation protocol. Note that one of the bimodal distributions used in our experiments is asymmetric, i.e., its modes have different values of standard deviation. We repeated the generation protocol ten times and accounted for the mean results, varying at each time the random initialization.

Table 10 – Description of the simulation protocol utilized in our experiments.

Property description	**Property value**
Sample size	50, 100, 200, 400, 500, 750, 1000, 2500, 5000, 7000, 10000, 15000, 25000, 50000, 75000, 100000, 200000, 500000, 1000000
Sampling distribution	Uniform, Normal, or Bimodal
Target function	Linear (`lin`) or Cubic (`cub`)
Amount of noisy instances	0% or 10%
Noise characteristics	$\mathcal{N}(0,0.1)$ or $\mathcal{N}(0,0.01)$[a]
Distribution name	**Distribution property**
Normal	$\mathcal{N}(0,1)$, $\mathcal{N}(0,0.1)$, $\mathcal{N}(0,7)$
Uniform	$[-1,1]$, $[-0.1,0.1]$, $[-7,7]$
Bimodal[b]	$\mathcal{N}(-1,1)\|\mathcal{N}(1,1)$, $\mathcal{N}(-0.1,0.1)\|\mathcal{N}(0.1,0.1)$, $\mathcal{N}(-7,7)\|\mathcal{N}(7,0.1)$

[a]Depending on the parameters of the generating distribution. We added normally distributed noise with smaller standard deviation to distributions whose dispersion was also small.
[b]We constructed bimodal distributions by sampling from two Normal distributions with equal probability. We use the "|" symbol to indicate a concatenation operation.

4.6.2 Settings used in the Attribute Observers

E-BST does not have hyperparameters to setup. TE-BST, on the other hand, was configured to truncate the input values to three decimal places. We evaluated three variants of QO to evaluate how r impacts on the obtained results, namely:

- $QO_{0.01}$: uses a fixed value (0.01) to discretize the input features.
- $QO_{\sigma \div 2}$: uses the standard deviation of the feature divided by 2 as the quantization radius.
- $QO_{\sigma \div 3}$: uses the standard deviation of the feature divided by 3 as the quantization radius.

Although the standard deviation of the whole monitored sample is not available beforehand in real-world scenarios, we can rely on incremental variance estimators. These approximations will be used when applying QO to ODT algorithms. Note that ODT regressors already keep one incremental variance estimator per leaf node to enable split candidate search. A fixed radius value, such as 0.01, can be applied at the beginning of the tree construction as a cold-start choice. Hence, we add the variant that uses a fixed radius in our experimental setup. QO is going to be integrated into `river`[1], a popular framework for online machine learning.

4.6.3 Evaluation metrics

We selected three performance evaluation metrics to evaluate the AOs. They measure how accurate were the splits, how much memory the AOs used, and how long the AOs took to process the data and evaluate the best split candidate.

The first measure was the split merit yielded by each AO, i.e., the obtained VR value. We also calculated the number of elements stored by each AO to estimate their memory usage. By element, we mean the number of nodes (E-BST and TE-BST) or the number of hash slots (QO). Since all the AOs store the same set of target statistics, we can rely on the number of elements rather than precisely measuring their actual memory usage. The time measurements were twofold: we measured the time taken by the AO to monitor the whole sample and the time they spent to produce a split candidate at the end of the stream monitoring.

For all the metrics, the smaller the value, the better. Besides, among all the metrics, the time measurements are the only ones that have a well-defined scale, i.e., they were measured in seconds.

4.7 Results and discussion

In Figure 8, we summarize, separately for the tasks `lin` and `cub`, the average results obtained in the experiments. We also created separate plots for each data distribution and

[1] <https://riverml.xyz>

regression task. However, looking at the experimental results, we observed that the differences in performance between the AOs followed similar patterns, regardless of the data distribution. Thus, we only present the charts of the averaged results.

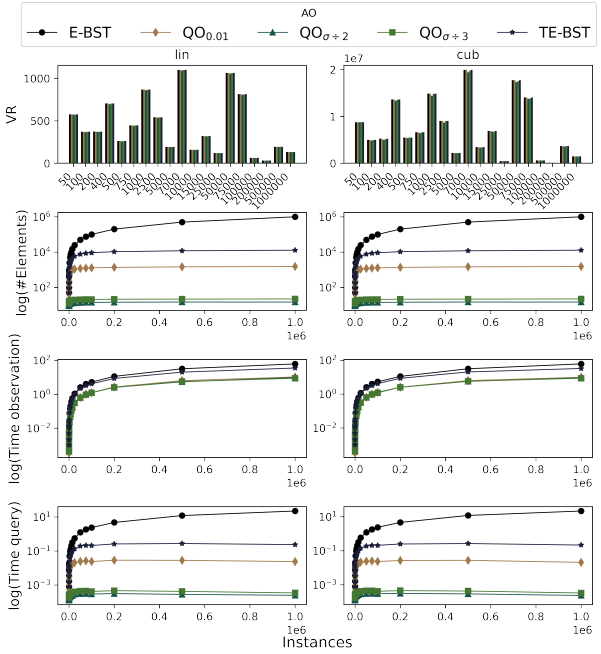


Figure 8 – Average results obtained by the compared AOs in the `lin` and `cub` tasks. From top to bottom: VR, and the logarithm of the number of stored elements, observation time (in seconds), and query time (in seconds).

In the next sections, we discuss each evaluation metric separately and refer to specific characteristics observed in Figure 8. Differently from Figure 8, when performing our in-depth analysis of each metric, we did not average the results between the different sampling data distributions. Instead, we accounted for the results obtained by the AOs, considering each evaluated sample size, data distribution, and regression task. We relied on Friedman tests and Nemenyi post-hoc tests (DEMŠAR, 2006) (with $\alpha = 0.05$) to evaluate the statistical significance of the different performances obtained by AOs.

4.7.1 Merit

As expected, the exhaustive (or near exhaustive) methods presented the highest VR values. E-BST and TE-BST consistently surpass the QO variants when it comes to VR, as shown in our statistical test (Figure 9). Nonetheless, the actual obtained VR values were very similar, regardless of the AO. We highlight this fact in the top portion of Figure 8, where the average VR obtained in tasks `lin` and `cub` is presented. It can be seen that the bars representing different AOs' split merits are similar for equal sample size, even considering that VR has a squared operation in its formulation, i.e., an operation that stretches the output range of the heuristic.

In Figure 10, we compare the differences between the split points obtained by TE-BST and QO against the E-BST ones. We observe that as we decrease QO's quantization radius, its

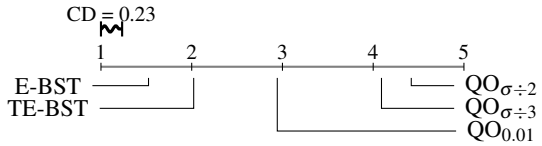


Figure 9 – Friedman test and Nemenyi post-hoc test when comparing the merit of the splits (VR) generated by the different AO algorithms ($\alpha = 0.05$).

splits become closer to those estimated by E-BST. The radius is directly correlated with both the obtained merit and the AO size/runtime: the smaller the radius, the higher the merit; similarly, the larger the radius, the smaller the runtime and memory usage.

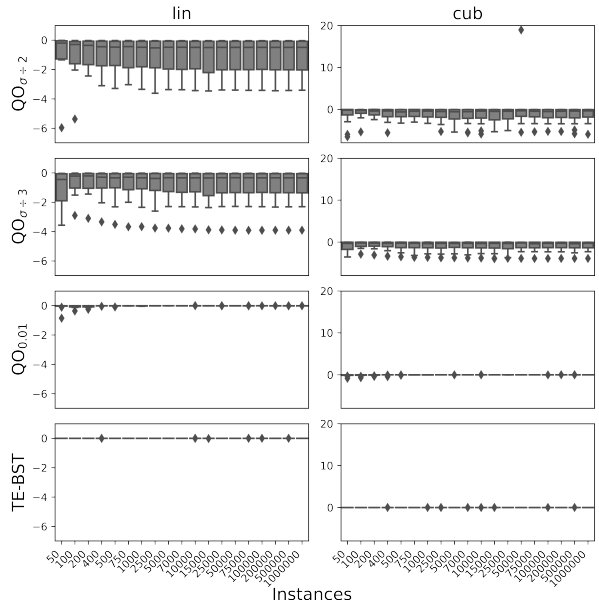


Figure 10 – Average differences between the split points found by QO and TE-EBST in comparison with E-BST.

4.7.2 Number of elements

The amount of memory used by an AO is directly proportional to the number of elements carried by it. In this study, element means a slot (QO) or node (E-BST/TE-BST) carrying a split value and target statistics.

The QO variants obtained the best rankings in this analysis. The reader might refer to the second row of Figure 8 to see how massive the differences were. The results are presented in log-scale so that we can visually compare the AOs. Without this visualization trick, the QO results would not be visible in the chart since our proposal used significantly less memory than its competitors.

The performed statistical test (Figure 11) also highlights the differences in memory usage. As expected, the larger the quantization radius, the more reduced is QO's memory footprint. In our experimental setup, the fixed radius $r = 0.01$ was smaller than the dynamical choices

($\sigma \div 2$ and $\sigma \div 3$). Hence, QO with a constant quantization radius spent more memory than the other variants. On the other hand, concerning split merit, the $QO_{0.01}$ was also the most accurate variant. Users might use different proportions of the feature's standard deviation to balance computational costs and the VR. Lastly, TE-BST stored significantly fewer elements than E-BST, as expected.

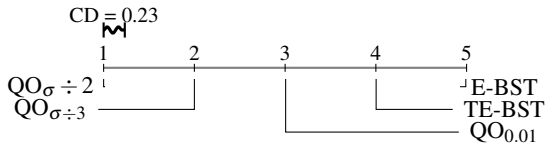


Figure 11 – Friedman test and Nemenyi post-hoc test when comparing the number of elements stored by the different AO algorithms ($\alpha = 0.05$).

4.7.3 Time

We divide the running time analysis into two parts: observation and query costs. When observing instances, all QO variants performed better than the E-BST variants, as illustrated in the third row of Figure 8. The differences between the QO variants were minimal in this analysis. Interestingly, although a smaller quantization radius resulted in increased memory usage, sometimes we observed that it also resulted in faster insertions. In other words, $QO_{\sigma \div 3}$ was faster than $QO_{\sigma \div 2}$ for monitoring data. Figure 12 illustrates this phenomenon. We hypothesize that sometimes the addition of new elements in a hash might be faster than handling collisions. There is, however, a delicate balance between the radius, memory usage, and running time. As the radius decreased more, the insertions became slower since testing for membership of a hash code also has a cost. In fact, $QO_{0.01}$ was the slowest variant of our proposal.

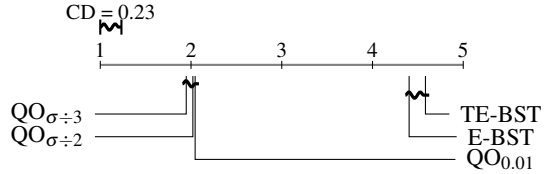


Figure 12 – Friedman test and Nemenyi post-hoc test when comparing the time spent by the different AO algorithms to monitor the input data ($\alpha = 0.05$).

Surprisingly, E-BST was generally faster than TE-BST. It might be related to the same situation of handling "collisions". In some cases, updating the statistics of existing nodes in the TE-BST is more time consuming than creating a new node. Nonetheless, the actual time differences observed between AO variants bearing from the same base algorithm were minimal. Hence, they might not have a high impact on real-world applications. The speed-up gains obtained by QO over E-BST (and TE-BST) were, however, clear.

When it comes to querying split points, QO variants were clearly faster than the E-BST variants. The bottom row of Figure 8 and Figure 13 illustrate how pronounced the differences were.

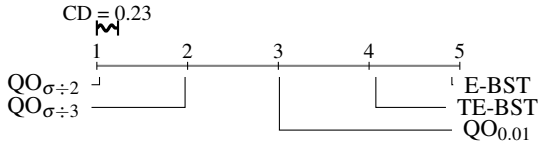


Figure 13 – Friedman test and Nemenyi post-hoc test when comparing the time spent by the different AO algorithms to query for split candidates ($\alpha = 0.05$).

$QO_{\sigma \div 2}$ performed faster than $QO_{\sigma \div 3}$ and $QO_{0.01}$, as it had fewer points to process. When querying for split points, the higher the number of stored slots, the slower is the processing. Finally, TE-BST was faster than E-BST to query split points, as we expected.

4.8 Final considerations

This paper introduced an efficient and effective algorithm for monitoring numerical input features in online regression trees. Our proposal, QO, requires significantly less memory and processing time than the current strategy used in practical applications. Moreover, QO can provide accurate split point suggestions by relying on an approximate algorithm rather than a greedy approach.

The experimental results suggest that QO could be easily integrated within online regression decision tree frameworks, such as Hoeffding Trees. In future works, we intend to evaluate the impact of using QO as attribute observers of such trees. QO can also be easily extended to deal with multi-target regression. We also intend to seek better alternatives to provide split candidate suggestions. One possible strategy to follow is the usage of meta-learning to recommend the split points. QO could be used to monitor data and provide the information necessary to induce meta-learning split point recommenders.

Acknowledgements

This work was supported by São Paulo Research Foundation – FAPESP (grant #2018/07319-6).

CHAPTER

5

FAST AND LIGHTWEIGHT BINARY AND MULTI-BRANCH HOEFFDING TREE REGRESSORS

Publication information: This chapter is an article published at a workshop of the IEEE International Conference on Data Mining.

Reference: MASTELINI, Saulo Martiello et al. Fast and lightweight binary and multi-branch Hoeffding Tree Regressors. **In: 2021 International Conference on Data Mining Workshops (ICDMW).** © IEEE, 2021. p. 380-388. Reprinted, with permission, from the authors.

5.1 Abstract

Incremental Hoeffding Tree Regressors (HTR) are powerful non-linear online learning tools. However, the commonly used strategy to build such structures limits their applicability to real-time scenarios. In this paper, we expand and evaluate Quantization Observer (QO), a feature discretization-based tool to speed up incremental regression tree construction and save memory resources. We enhance the original QO proposal to create multi-branch trees when dealing with numerical attributes, creating a mix of interval and binary splits rather than binary splits only.

[*] <http://www.ieee.org/publications_standards/publications/rights/rights_link.html>

We evaluate the multi-branch and strictly binary QO-based HTRs against other tree-building strategies in an extensive experimental setup of 15 data streams. In general, the QO-based HTRs are as accurate as traditional HTRs, incurring one-third of training time at only a fraction of the memory resource usage. The obtained numerical multi-branch HTRs are shallower than the strictly binary ones, significantly faster to train, and they keep predictive performance similar to the traditional incremental trees.

5.2 Introduction

Decision trees (DT) are ubiquitous. These supervised Machine Learning (ML) models have been a popular choice among researchers and practitioners throughout the years. This observation is valid concerning traditional *in-batch* or static ML (FERNÁNDEZ-DELGADO *et al.*, 2014; CHEN; GUESTRIN, 2016; KE *et al.*, 2017) and even more pronounced in stream mining and online learning (DOMINGOS; HULTEN, 2000; GAMA, 2010; IKONOMOVSKA; GAMA; DŽEROSKI, 2011b; IKONOMOVSKA; GAMA; DŽEROSKI, 2015; BIFET *et al.*, 2018).

Different from static DTs, the incremental ones start their building with only one (leaf) node. The structure is progressively expanded as data is processed. Historically, Hoeffding Trees (HT) have been the most popular family of incremental DTs for stream mining (DOMINGOS; HULTEN, 2000; IKONOMOVSKA; GAMA; DŽEROSKI, 2011b; BIFET *et al.*, 2018). HTs use the Hoeffding bound to determine when enough data was observed to enable splits.

This family of incremental ML algorithms was developed to deal with situations where data arrives incrementally and can be potentially unbounded (BIFET *et al.*, 2018). The input data is in the form $(\vec{x}, y)_{t=0}^{\infty}$, where $\vec{x}$ and y represent realizations of the feature vector and the target at time step t, respectively. When y belongs to a set of unordered and discrete values, the resulting ML task is called classification. When $y \in \mathbb{R}$, the resulting task is called regression.

Traditionally, classification HTs received more attention from the research community than regression trees (IKONOMOVSKA; GAMA; DŽEROSKI, 2011b; GOMES *et al.*, 2018). Still, HT regressors (HTR) are powerful algorithms for dealing with massive datasets in an incremental learning fashion. HTRs grow by creating partitions that minimize the variance in y. Although data can arrive indefinitely, the amount of computational resources is limited (GAMA, 2010; BIFET *et al.*, 2018). Thus, efficient data structures to keep the variance in y for different partitions applied to each feature in $\vec{x}$, are needed. Partitions in numerical features have been historically limited to binary tests in both incremental and non-incremental DTs.

Due to the lack of research, for many years, the applicability of HTRs was bounded by the high costs involved in dealing with numerical attributes (IKONOMOVSKA; GAMA; DŽEROSKI, 2011b; GOMES *et al.*, 2020; MASTELINI; CARVALHO, 2021). The commonly used strategy to evaluate split decisions in HTRs works similarly to the non-incremental re-

gression DTs (IKONOMOVSKA; GAMA; DŽEROSKI, 2011b; DUARTE; GAMA; BIFET, 2016; GOMES *et al.*, 2018). In fact, all the observed input points (IKONOMOVSKA; GAMA; DŽEROSKI, 2011b) or a subset of them (DUARTE; GAMA; BIFET, 2016) are explicitly stored by the HTRs. This choice can render the HTR building time and memory usage impractical in real-time applications.

In (MASTELINI; CARVALHO, 2021), the authors introduce an effective feature discretization-based mechanism to build HTRs. Their proposal improves upon the traditional strategy (IKONOMOVSKA; GAMA; DŽEROSKI, 2011b) in terms of memory and running time. This new strategy, called Quantization Observer (QO), was only evaluated in synthetic scenarios and not directly applied to HTRs. In this paper we extend (MASTELINI; CARVALHO, 2021) in two directions: (1) we evaluate how QO performs when applied to HTRs; (2) we take advantage of the numerical feature quantization nature of QO to create multi-branch HTRs.

By using an extensive experimental setup comprising of multiple datasets, evaluation metrics, and HTR-building algorithms, we explore multiple hyperparameter values for QO and its competitors in order to answer the following research questions:

Q.1: Is QO able to deliver trees with competitive predictive performance while also speeding up model construction and reducing memory usage?

Q.2: Are the QO's multi-branch enabled HTRs appealing in terms of predictive performance and computational resource usage?

Q.3: Are numerical multi-branch trees more compact than the strictly binary ones in terms of structure, and thus possibly easier models for manual inspection?

We answer the presented questions throughout the paper. The remainder of this work is organized as follows: section 5.3 presents related work and contextualize HTR-building strategies. In section 5.4 we show how numerical multi-branch HTRs can be constructed with QO. Section 5.5 describes the experimental setup used to compared the different compared incremental regression models and answer our research questions. Next, we present and discuss the obtained results in section 5.6. Lastly, we present our final considerations, open issues, and possible directions for further research in section 5.7.

5.3 Related Work

The seminal Fast Incremental Model Tree with Drift Detection (FIMT-DD) (IKONOMOVSKA; GAMA; DŽEROSKI, 2011b) adapted from incremental classification tasks the concept of using binary search trees to keep attribute-target statistics. It paved the way to build incremental regression trees and rule-based regression models by defining the Extended Binary Search Tree (E-BST) data structure.

E-BST belongs to a family of algorithms/data structures called attribute observers (AO)(BIFET *et al.*, 2018), in charge of monitoring the input values and their relationship with the target variable. HTs carry one AO instance per attribute at each one of their leaves, i.e., each AO instance monitors one input feature at the time.

As described in (IKONOMOVSKA; GAMA; DŽEROSKI, 2011b) and (MASTELINI; CARVALHO, 2021), E-BST carries an incremental variance estimator at each one of its nodes. Each node represents a distinct monitored input value. By combining mathematical expressions to merge and divide partial variance estimates, each stored input value can be used as a split candidate. Therefore, for each stored v value we can estimate the variance of elements smaller than or equal to v, and the elements greater than v. These tests are the split candidates typically evaluated by HTRs in numerical features. Although E-BST-based trees are incremental, they function similarly to batch DTs.

Unfortunately, E-BST has a $O(\log n)$ cost per insertion and a $O(n)$ cost for evaluating split candidates. In the worst case, insertions might be $O(n)$. The memory cost of E-BST is $O(n)$. In the expressions, n is the number of instances stored in the E-BST structure. If a split attempt succeeds, the existing AO instances in the leaf node are discarded, and new instances are created in the descendant nodes. If no split is performed, E-BST keeps gathering more input values, and the computational cost might become prohibitive.

As alternatives, one can limit the number of stored points (DUARTE; GAMA; BIFET, 2016) or round the values to a given number of decimal places before adding them in the E-BST (MASTELINI; CARVALHO, 2021). Multiple regression DT-based (IKONOMOVSKA; GAMA; DŽEROSKI, 2015; OSOJNIK; PANOV; DŽEROSKI, 2017; OSOJNIK; PANOV; DŽEROSKI, 2018; OSOJNIK; PANOV; DŽEROSKI, 2020), rule-based (SOUSA; GAMA, 2018), and ensemble-based (GOMES *et al.*, 2018; GOMES *et al.*, 2020) solutions proposed in the past years rely on E-BST or one of its variants to build the learning models. Another alternative employed in the literature is the usage of a calibration set to define static split candidate points (GOUK; PFAHRINGER; FRANK, 2019) that will be used during the whole processing. While doable, this alternative does not cope with the dynamic characteristics of evolving data streams.

In (MASTELINI; CARVALHO, 2021) the authors proposed the Quantization Observer (QO), a new AO algorithm for regression that reduces the costs involved in performing splits in numerical features. QO has a $O(1)$ cost per insertion, and $O(h \log h)$ and $O(h)$ of split evaluation and memory costs, respectively. In the cost expressions, h represents the number of slots stored in the QO hash-like structure. Usually, $h \ll n$. Our proposal is rooted in QO. In the next section we briefly describe the QO algorithm and how it can be applied to create multi-branch regression trees.

5.4 Fast multi-branch enabled numerical attribute splits

Instead of storing all the observations as in E-BST, QO quantizes the incoming data by creating equal-spaced partitions in the feature values. To do so, a fixed radius parameter $r > 0$ is applied in the following quantization rule to obtain a hashing code h_c for an observation x of the i-th input feature x_i:

$$h_c = \left\lfloor \frac{x}{r} \right\rfloor . \tag{5.1}$$

The h_c value determines the slot position x will be mapped to and aggregated in a hash-like structure. QO's slots store a mean estimator of x_i along with a variance estimator of y. By relying on the formulae presented in (MASTELINI; CARVALHO, 2021), one can manipulate the statistics of the monitored segments to estimate the split statistics for the whole monitored sample.

In the original QO proposal, only binary splits are attempted. In other words, given two consecutive QO slots (w.r.t. their mean x_i value), an intermediate threshold is evaluated, and from this split, two branches are created. The left branch statistics are calculated by aggregating all slots' statistics before the threshold. The remaining statistics (right branch) can be obtained by subtracting the left branch statistics from a variance estimator computed for the whole sample. Thus, binary splits can be evaluated at reduced costs in comparison with the usage of E-BST.

Using synthetic experiments, QO's authors showed their proposal surpasses two variants of E-BST: the vanilla E-BST and a variant that rounds the incoming data to d decimal places before adding the instances to the binary search tree. This variant is referred to as Truncated E-BST (TE-BST). In this paper, we demonstrate the application of QO in the HTRs and evaluate its performance against E-BST and TE-BST.

QO also enables the exploration of a promising and distinct way to build HTRs. Each QO slot stores split-enabling statistics concerning a fixed-length interval of the monitored input feature. Such intervals, which are proportional to r in length, can be mapped to tree branches. Hence, instead of applying a single binary split test $x_i \leq v, v \in \mathbb{R}$ and its complement $x_i > v$, we divide the input feature into equal-sized intervals. The resulting split tests have the form $x_i \in [a, b)$, where $a = h_c \times r$ and $b = a + r$. Actually, in practice, incoming x_i values are mapped to their respective branches by using (5.1). By storing the used quantization radius, we can extrapolate the currently observed intervals and create new branches if samples belonging to unobserved portions of the feature space arrive.

Therefore, we can create n-ary trees rather than binary ones. The number of created branches depends on the spread of the feature's values and the chosen r. Such n-ary trees ought to be wider than the strictly binary ones, although shallower. Those shallower trees may be easier to inspect and interpret visually, besides having reduced training costs. We hypothesize

that although wider, such trees ought to perform fewer splits when compared to the strictly binary ones. Hence, one could potentially decrease the split attempt frequency, which is a costly operation and still retain similar predictive performance to the traditional HTRs. Note that both binary and multi-branch split candidates can be jointly evaluated. In this case, the resulting trees would have a mix of binary and multi-branch splits.

As a downside, choosing an inadequate r value can lead to the creation of too many branches and consequently many leaves. Besides making the multi-branch trees require more memory than the strictly binary ones, surplus leaves might also decrease the predictive performance. With fewer instances reaching the leaves, the tree's predictions can become inaccurate as the leaf predictors will be rarely updated.

Accordingly, in our experiments, we explore different values of r in both strictly-binary HTRs and multi-branch enabled ones to identify promising candidates.

5.5 Experimental setup

In this section we present our experimental setup concerning datasets, evaluation metrics and strategy, and compare incremental learning algorithms. We performed all experiments using the River (MONTIEL *et al.*, 2021) online learning library[2] in a machine with an Intel Xeon Silver 4114 CPU at 2.20GHz, 128 Gigabytes of RAM, and running Debian 9.13.

5.5.1 Benchmark datasets

We summarize the selected datasets' characteristics in Table 11. They represent commonly used data sources from the incremental regression literature coming from diverse real and synthetic domains. Datasets marked with "*" are synthetic.

5.5.2 Regression tree variants

In this study, we do not consider model trees, i.e., models that have a decision tree structure but can use different kind of learning models in their leaves. We limit our analysis to "pure" regression trees whose leaves predict the target mean. This decision is twofold: firstly, some of the evaluated datasets have nominal features, which cannot be easily manipulated by linear regression models, the popular choice for model trees' leaves. Secondly, we wanted to avoid external factors that might shadow differences between the performances of different tree variants. We focus on the tree structure, rather than on its prediction phase's specifics.

We compared HTRs when using E-BST, TE-BST, and QO as AO algorithms. In the QO case, we also considered the possibility of evaluating multi-branch splits among the binary ones. The applied HTRs are similar to FIMT-DD except that they do not have concept drift

[2] <https://riverml.xyz>

Table 11 – Characteristics of the evaluated datasets.

Dataset	#Instances	#Numeric features	#Nominal features
Abalone	4977	7	1
Ailerons	13750	40	0
Bike	17379	12	0
CalHousing	20500	8	0
Elevators	16599	18	0
House8L	22784	8	0
House16H	22784	16	0
Metro	48204	4	3
Pol*	15600	48	0
Wind	6574	12	2
Wine	5298	11	0
Friedman*	100000	10	0
MV*	100000	7	3
Puma8NH*	8192	8	0
Puma32H*	8192	32	0

adaptation capabilities. Nonetheless, such an adaptive aspect of FIMT-DD and other adaptive HT models (BIFET; GAVALDÀ, 2009; MANAPRAGADA; WEBB; SALEHI, 2018) are independent of the AOs. QO-based adaptive HTRs can be trivially assembled. This kind of exploration is, however, out-of-the-scope of this work.

Table 12 introduces the HTR variants compared in this study and their evaluated hyperparameters. We refer to the QO version that evaluates both binary and n-ary splits as QO^M. When no superscript is present, only binary splits are considered. On the other hand, we indicate QO's r values as subscripts.

Table 12 – Compared attribute observer algorithms and their hyperparameter values.

AO	Hyperparameters
E-BST	–
TE-BST	$d = 2$
QO	$r \in \{0.1, 0.25, 0.5, 1\}$
QO^M	$r \in \{0.1, 0.25, 0.5, 1\}$

The remaining HTR hyperparameters were kept to their default values, as commonly used in the literature and implemented in River (MONTIEL *et al.*, 2021).

5.5.3 Evaluation setup

We relied on the test-then-train (prequential) evaluation strategy in all the cases (GAMA, 2010). In this paper, we limit our analysis to stationary streams (without concept drift). Although concept drift is a pivotal concern in online learning, it does not affect the HTR building mechanics per se. In this stage, we are primarily concerned with comparing the potential of the different DT

structures and their building strategy. However, multi-branch HTRs might be affected by concept changes differently than the strictly binary ones. Combining multi-branch HTRs and concept drift countermeasures (BIFET; GAVALDÀ, 2009) remains an open issue for future investigation.

Given that the benchmark streams are stationary, we performed a pseudo shuffle of the datasets' instances using a Reservoir Sampling technique with a window of 100 instances. We report the results obtained by averaging five executions using different random seeds. No label delay was considered in this study.

In all the cases, we assembled an evaluation pipeline comprising of one feature scaler and one regressor. The chosen online scaler centered numerical features' values to their mean and scaled them to unit variance. HTRs can deal with nominal features automatically, by creating one tree branch per category value. However, the considered baselines (subsection 5.5.4) do not have this capability. Thus, baseline models used an slightly modified evaluation pipeline, where nominal features were one-hot-encoded. The source code used in our experiments can be consulted in <https://github.com/smastelini/icdm2021_multiway_splits>.

We measured the Root Mean Square Error (RMSE) of the compared models, memory usage (in Megabytes – MB) and running time (in seconds – s). Since we are comparing multiple regression models and datasets, we cannot guarantee that the observed performance differences are not a product of chance. Hence, we apply statistical tests to assess the performance differences' significance. As we do not make assumptions concerning the distribution of the measured performance metrics, we apply the non-parametric Friedman test ($\alpha = 0.05$) followed by the post-hoc Nemenyi test. The results are displayed using critical distance (CD) diagrams (DEMŠAR, 2006).

We also measured some statistics related to the tree structures. Namely, the tree height, the number of split (decision) nodes, leaves, and the total number of nodes (split + leaves).

5.5.4 Baselines

We ran Passive-Aggressive Regression (PAR) models (CRAMMER *et al.*, 2006) and a Dummy regressor to compare against the HTRs. PAR is a popular linear regression incremental learning algorithm applied to stream mining tasks. The Dummy regressor was defined as an incremental target mean estimator, i.e., it outputs the current mean estimation of y.

5.5.5 Case study

Aside from the datasets used to benchmark the HTR variants and baselines, we also performed a case study using records encompassing two years (2007-2008) of the Airlines dataset (IKONOMOVSKA; GAMA; DŽEROSKI, 2011b; DUARTE; GAMA; BIFET, 2016). This case study intended to evaluate how the different HTRs perform when facing a large number

of instances to process. The Airlines dataset has 13 attributes, 6 of them are numerical. The selected subset, Airlines07-08, has 13 085 750 instances.

We followed a slightly different evaluation protocol, due to the sheer size of this dataset and time limitations. Each selected HTR variant processed the Airlines07-08 dataset once (refer to subsection 5.6.1 for details on the selected HTRs). At intervals of one thousand instances we recorded the RMSE, total memory usage, and the total training and testing time of the compared models. We also collected data regarding the characteristics of the HTRs, namely: the number of nodes, split nodes, leaves, and the tree height.

5.6 Results

This section presents the obtained results when performing the steps described in section 5.5. We start by describing how the best QO variants were selected for further comparisons. Next, we compare the selected tree models with the baselines. Lastly, we present a case study using the Airlines dataset.

5.6.1 Selecting the best QO variants

When comparing algorithm variants over multiple performance metrics and datasets, selecting candidates that offer a good compromise between all metrics is not easy. Some metrics should not be analyzed independently when selecting the best algorithm variant, e.g., sometimes to reduce the prediction error implies increasing the processing time.

Nonetheless, we still want to find the best balance between predictive capacity and resource usage. In other words, we want the QO variants yielding the smallest RMSE values and using the minimum amount of computational resources (memory and processing time). We relied on a Principal Component Analysis (PCA) biplot (GABRIEL, 1971) to select the best QO variants among the ones considered.

PCA biplots depict both the placement of the transformed data and the direction of the original dimensions projected in a two-dimensional visualization. Thus, we can visually inspect the interactions between performance metrics and the compared models. However, this chart should not be used as the sole source of information to assess the performance of the HTR models. Although the biplot is a useful visual aid, one should consult the actual experimental performance measurements for detailed comparisons.

To build the biplot, we followed the steps:

1. For each dataset, we normalized each metric value of all QO variants between 0 and 1. For all the considered metrics (RMSE, memory usage, and running time), the smaller the value, the better. Thus, the worst-performing variant obtained 1, whereas the best one

Table 13 – RMSE results.

Dataset	PAR	Dummy	HTR + E-BST	HTR + TE-BST	HTR + $QO_{0.25}$	HTR + QO_1^M
Abalone	5.5440 ± 0.05	3.3412 ± 0.00	3.0944 ± 0.03	3.0779 ± 0.01	3.1511 ± 0.01	**2.7823 ± 0.09**
Ailerons	1.0216 ± 0.02	0.0004 ± 0.00	0.0003 ± 0.00	**0.0003 ± 0.00**	0.0003 ± 0.00	0.0003 ± 0.00
Bike	140.4632 ± 0.33	181.5973 ± 0.02	**119.8733 ± 5.56**	126.1380 ± 6.90	143.2146 ± 2.61	142.7532 ± 1.54
CalHousing	226188.3564 ± 15.86	115229.2171 ± 2.94	85907.8297 ± 1301.72	86408.6345 ± 1739.93	**85164.6955 ± 831.31**	90327.9694 ± 2747.15
Elevators	0.9345 ± 0.01	0.0067 ± 0.00	**0.0053 ± 0.00**	0.0053 ± 0.00	0.0058 ± 0.00	0.0056 ± 0.00
House8L	65898.3592 ± 32.82	52868.1757 ± 37.97	**40626.4686 ± 760.29**	40736.0214 ± 393.84	41244.3914 ± 1984.34	42998.9093 ± 1617.35
House16H	66064.4393 ± 32.26	52868.1757 ± 37.97	**44071.0916 ± 880.17**	44132.8165 ± 804.91	45975.6889 ± 143.04	47833.6526 ± 82.53
Metro	1999.7060 ± 4.39	1987.2024 ± 0.06	1961.2533 ± 4.18	1958.0165 ± 7.02	1958.4314 ± 4.35	**1949.0318 ± 0.64**
Pol	39.1908 ± 0.10	41.8120 ± 0.01	**24.7070 ± 0.84**	24.8886 ± 0.75	25.1798 ± 1.51	30.0638 ± 2.69
Wind	9.6295 ± 0.03	6.6900 ± 0.01	4.9791 ± 0.04	4.9941 ± 0.05	4.9447 ± 0.04	**4.8229 ± 0.20**
Wine	4.4754 ± 0.03	0.8957 ± 0.00	**0.8084 ± 0.02**	0.8107 ± 0.01	0.8294 ± 0.01	0.8297 ± 0.01
Friedman	7.5243 ± 0.01	4.9826 ± 0.01	**2.5815 ± 0.04**	2.6065 ± 0.05	2.7864 ± 0.03	2.6108 ± 0.09
MV	21.8740 ± 0.06	30.6621 ± 0.10	8.7728 ± 0.64	10.0263 ± 1.06	**8.6413 ± 0.72**	9.8663 ± 0.39
Puma8NH	7.6525 ± 0.05	5.6271 ± 0.00	4.0372 ± 0.02	4.0234 ± 0.01	4.2829 ± 0.05	**3.7767 ± 0.01**
Puma32H	0.7887 ± 0.01	0.0303 ± 0.00	0.0191 ± 0.00	0.0193 ± 0.00	0.0228 ± 0.00	**0.0180 ± 0.00**
Avg. rank	5.67	5.20	**1.87**	2.33	3.20	2.73
Avg. rank real	5.64	5.18	**1.82**	2.18	3.18	3.00
Avg. rank synth.	5.75	5.25	**2.00**	2.75	3.25	**2.00**

received 0. The remaining variants received values between 0 and 1 and proportional to their performance.

2. We took the mean value of each metric among all the datasets as a representative value of each QO variant. Hence, three values of mean normalized performance represented each QO variant.

3. A two-dimensional PCA transformation was applied to the values obtained in the previous step. The biplot chart used this data transformation.

In the obtained chart, QO variants are represented as points, and the performance metrics are depicted as vectors (arrows) starting from the origin. The vectors' directions indicate the direction in which the metric values increase. The direction also indicates correlation. For instance, vectors pointing in the same direction are an indication that in the experimental results, the metrics are positively correlated. The metrics are negatively correlated if two metric arrows point to opposite directions (one increases while the other decreases, and vice-versa). If the vectors are approximately orthogonal, the metrics they represent are not strongly correlated.

The placement of the points indicates how biased they are regarding each metric. For instance, points placed close to a vector extremity indicate that the corresponding QO variants tend to yield high values in the analyzed metric. If points are placed in the opposite direction of a vector, then the QO variants tend to obtain low values in the corresponding performance metric.

Our experimental findings are presented in Fig. 14. It is worth noting that the scale is not meaningful in the figure since we performed multiple normalization and summarization steps. Instead, we are interested in the placement of the elements in the chart.

We can observe that memory usage and running time are strongly correlated. Hence, the higher the memory usage, the slower is the tree model. This observation was expected because bigger tree structures take more time to traverse. Similarly, QO instances where a small

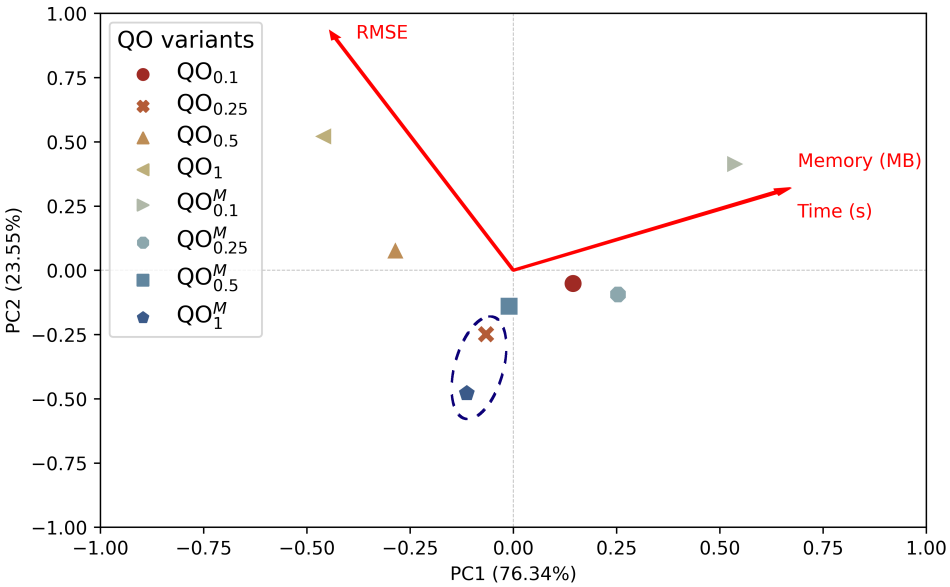


Figure 14 – PCA biplot comparing QO variants (displayed as dots) w.r.t. performance metrics (displayed as vectors indicating the growth direction for a metric). The placement of a model (point) indicates how biased it is towards a metric. Metrics inside the blue circle show the best compromise between predictive performance and computation resource usage.

discretization radius is used end up storing an increased number of hash slots. Hence, both the strictly binary trees and multi-branch ones that rely on small quantization radii resulted in increased computational resource usage. The impact of the radius value is more pronounced on the multi-branch trees as the number of created branches after each split is directly related to the quantization radius.

On the other hand, the final RMSE did not appear strongly correlated to the variants' computational resource usage. Still, the worst performers (the points placed along the RMSE vector's direction in Fig. 14) were also the QO variants with strictly binary splits and the highest radius values (QO_1 and $QO_{0.5}$). Interestingly, creating too many tree branches did not seem to result in RMSE improvements: $QO^M_{0.1}$ was the most computational resource-intensive variant, while also being one of the worst contenders regarding RMSE. We hypothesize that only a few instances end up reaching each leaf since there are too many partitions. Therefore, the obtained predictions are not accurate.

We highlight in Fig. 14 the two QO variants that presented the best balance between predictive performance and computation resource usage. Both $QO_{0.25}$ and QO^M_1 are placed in the opposite direction of the metrics' vectors in the chart. Thus, they represent the QO variants that most effectively jointly minimize RMSE, memory usage, and running time. From here onward, we will focus on these two variants in our comparisons.

Table 14 – Memory (MB) results.

Dataset	PAR	Dummy	HTR + E-BST	HTR + TE-BST	HTR + $QO_{0.25}$	HTR + QO_1^M
Abalone	0.0202 ± 0.00	**0.0188 ± 0.00**	2.9680 ± 0.41	1.5880 ± 0.23	0.4407 ± 0.01	0.3517 ± 0.05
Ailerons	0.0266 ± 0.00	**0.0237 ± 0.00**	41.9340 ± 6.05	13.9920 ± 2.20	7.9200 ± 0.59	8.8240 ± 1.91
Bike	0.0201 ± 0.00	**0.0186 ± 0.00**	18.4140 ± 3.48	5.0680 ± 1.12	2.1160 ± 0.30	0.8842 ± 0.07
CalHousing	0.0184 ± 0.00	**0.0173 ± 0.00**	15.3700 ± 1.98	8.4240 ± 1.17	2.6800 ± 0.22	2.8000 ± 0.24
Elevators	0.0211 ± 0.00	**0.0194 ± 0.00**	25.3900 ± 4.77	8.4380 ± 0.51	5.0980 ± 0.44	3.7980 ± 0.93
House8L	0.0184 ± 0.00	**0.0173 ± 0.00**	20.6560 ± 0.64	10.6040 ± 1.13	3.7180 ± 0.35	3.2220 ± 0.19
House16H	0.0207 ± 0.00	**0.0191 ± 0.00**	37.1880 ± 1.43	21.2620 ± 1.69	9.2280 ± 0.53	7.1480 ± 0.72
Metro	0.0316 ± 0.00	**0.0229 ± 0.00**	5.4960 ± 1.14	1.2756 ± 0.20	1.0529 ± 0.20	0.9715 ± 0.11
Pol	0.0318 ± 0.00	**0.0276 ± 0.00**	32.2800 ± 4.18	7.9400 ± 1.03	7.1000 ± 0.58	6.4160 ± 1.55
Wind	0.0305 ± 0.00	**0.0238 ± 0.00**	22.0040 ± 0.89	14.9580 ± 0.61	4.0980 ± 0.02	1.4004 ± 0.75
Wine	0.0200 ± 0.00	**0.0186 ± 0.00**	6.5460 ± 0.62	4.0080 ± 0.54	1.7180 ± 0.21	1.0856 ± 0.25
Friedman	0.0184 ± 0.00	**0.0173 ± 0.00**	110.2500 ± 6.08	78.8420 ± 2.26	19.8680 ± 0.81	14.7340 ± 1.19
MV	0.0200 ± 0.00	**0.0181 ± 0.00**	12.1280 ± 1.71	7.9580 ± 1.75	2.8080 ± 0.13	2.4980 ± 0.27
Puma8NH	0.0184 ± 0.00	**0.0173 ± 0.00**	8.3980 ± 0.91	6.4260 ± 0.79	1.8960 ± 0.16	0.6024 ± 0.00
Puma32H	0.0252 ± 0.00	**0.0226 ± 0.00**	30.8800 ± 5.87	21.3500 ± 3.77	7.0820 ± 0.53	1.9980 ± 0.08
Avg. rank	2.00	**1.00**	6.00	5.00	3.87	3.13
Avg. rank real	2.00	**1.00**	6.00	5.00	3.82	3.18
Avg. rank synth.	2.00	**1.00**	6.00	5.00	4.00	3.00

5.6.2 Hoeffding Trees against other baselines

We compare the chosen HTR variants against the baselines defined in section 5.5. In Table 13 we present the results concerning RMSE. In general, E-BST-equipped HTR obtained the smallest RMSE values in both real-world data and synthetic datasets. This observation was expected since this AO operates similarly to the non-incremental tree regressors. QO_1^M obtained the same ranking of E-BST regarding the synthetic datasets. TE-BST appears in the second general ranking, followed by $QO_{0.25}$. Interestingly, the Dummy predictor performed slightly better than PAR.

Regarding memory usage (Table 14), as we expected, the baselines used the smallest amount of memory resources. Among the HTRs, however, both evaluated QO variants were the most memory-conservative AOs. Interestingly, QO_1^M obtained the best ranking, closed followed by $QO_{0.25}$. We argue that multi-branch HTRs are shallower than the strictly binary ones. Therefore, HTRs using QO_1^M were able to save memory resources because of the AO choice and their structure. The HTRs using $QO_{0.25}$ also benefit from memory savings but ought to be as deep as the ones using E-BST and TE-BST. As expected, E-BST was the most memory-intensive AO, followed by TE-BST. The rounding procedure applied to TE-BST reduces memory costs but does not surpass QO asymptotically, as our experiments demonstrated.

The running time results, presented in Table 15, follow a similar behavior to the memory usage. Such a result confirms the observations made using Fig. 14. The memory usage and the running time of HTRs are correlated.

The diagrams in Fig. 15 assess the significance of the obtained performance differences (top: RMSE, middle: memory, bottom: time). There were no significant differences between the HTRs' RMSE. All HTRs were significantly more accurate than the baselines in the benchmark datasets. HTRs using QO_1^M used significantly less memory resources than the ones using other AOs. QO_1^M-equipped HTRs used resources comparable to the baselines, according to the

Table 15 – Time (s) results.

Dataset	PAR	Dummy	HTR + E-BST	HTR + TE-BST	HTR + $QO_{0.25}$	HTR + QO_1^M
Abalone	1.2752 ± 0.07	**0.9173 ± 0.03**	8.7121 ± 1.26	4.3487 ± 0.04	1.3663 ± 0.01	1.1776 ± 0.03
Ailerons	7.3573 ± 0.03	**5.4067 ± 0.02**	185.8633 ± 18.83	57.9702 ± 8.15	24.4935 ± 1.38	24.6072 ± 3.11
Bike	4.0870 ± 0.02	**3.0097 ± 0.02**	73.3240 ± 5.16	24.4220 ± 2.98	9.4483 ± 0.69	6.2986 ± 0.12
CalHousing	4.1366 ± 0.15	**2.9193 ± 0.01**	59.9784 ± 5.66	37.3116 ± 1.89	10.9412 ± 0.34	11.6014 ± 0.92
Elevators	5.2953 ± 0.29	**3.6483 ± 0.01**	99.1738 ± 8.05	38.8695 ± 2.76	16.8792 ± 1.42	13.7311 ± 2.32
House8L	4.4784 ± 0.02	**3.2894 ± 0.01**	88.4930 ± 4.04	46.8992 ± 1.88	14.8069 ± 1.54	13.1912 ± 0.63
House16H	6.9056 ± 0.45	**4.6849 ± 0.01**	163.5159 ± 4.52	95.9075 ± 2.83	34.8241 ± 1.19	27.0146 ± 3.05
Metro	23.9037 ± 0.22	**13.9407 ± 0.03**	72.6021 ± 4.91	24.6162 ± 1.03	17.6904 ± 1.68	15.2675 ± 0.71
Pol	9.5710 ± 0.12	**6.7205 ± 0.01**	131.6997 ± 2.74	35.9275 ± 2.33	25.9872 ± 0.69	24.5118 ± 2.66
Wind	3.5233 ± 0.23	**2.1191 ± 0.03**	29.0415 ± 1.10	21.6813 ± 0.28	6.4846 ± 0.10	2.8160 ± 0.65
Wine	1.2386 ± 0.10	**0.8794 ± 0.01**	10.2921 ± 0.65	6.1503 ± 0.43	2.4771 ± 0.13	1.8703 ± 0.15
Friedman	21.7654 ± 1.04	**15.0029 ± 0.01**	1890.2800 ± 84.07	1359.9289 ± 48.97	282.3382 ± 11.75	174.5941 ± 19.93
MV	24.1791 ± 0.66	**16.7435 ± 0.04**	387.3366 ± 42.35	249.6484 ± 35.35	68.9306 ± 2.00	55.2576 ± 3.54
Puma8NH	1.5949 ± 0.01	**1.1712 ± 0.01**	15.6673 ± 0.42	11.8548 ± 0.58	3.5492 ± 0.08	2.3482 ± 0.02
Puma32H	3.7236 ± 0.01	**2.6284 ± 0.01**	62.4950 ± 1.30	51.1534 ± 3.39	12.9202 ± 0.34	7.3944 ± 0.08
Avg. rank	2.27	**1.00**	6.00	5.00	3.80	2.93
Avg. rank real	2.36	**1.00**	6.00	5.00	3.73	2.91
Avg. rank synth.	2.00	**1.00**	6.00	5.00	4.00	3.00

performed statistical tests. $QO_{0.25}$ also was statistically comparable to PAR regarding memory usage. Both the considered QO variants were significantly more efficient than E-BST, although $QO_{0.25}$ did not use significantly less memory resources than TE-BST. Similar conclusions can be drawn for the running time. However, the advantage of the QO variants over E-BST and TE-BST is not as pronounced as in memory usage results.

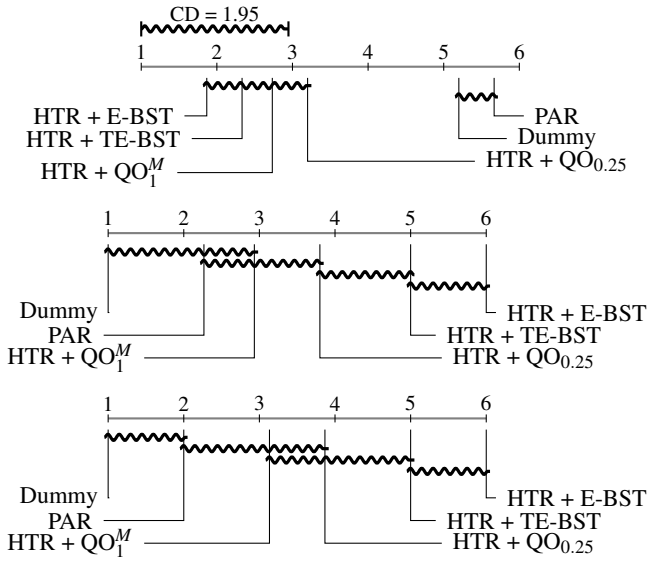


Figure 15 – Friedman test and Nemenyi post-hoc test analysis of the compared HTR models and baselines. Top: RMSE, middle: memory, bottom: time.

Therefore, regarding our first research question (**Q.1**), QO-based HTRs are competitive in terms of predictive performance while spending much less computation resources in comparison to E-BST and TE-BST. The same observation holds for the multi-branch HTRs derived from QO (research question **Q.2**).

Finally, we present statistics related to the compared HTRs in Table 16. As we had hypothesized (**Q.3**), QO_1^M generated shallower HTRs in comparison to the binary ones originated by E-BST, TE-BST, and $QO_{0.25}$. On the other hand, E-BST and TE-BST generated trees with a

reduced number of nodes compared to the QO ones. The number of decision nodes was generally smaller in QO_1^M in comparison to the other AOs due to the multi-branch tree architecture. The cases where this observation did not hold were probably due to a mix of binary and multi-branch splits. This is an open issue for further exploration and inspection. On the other hand, QO_1^M-based HTRs created more leaves than the strictly binary HTRs as a consequence of using multiple branches per split.

Table 16 – Structural characteristics of the compared Hoeffding Tree Regressors

Dataset	Metrics	HTR + E-BST	HTR + TE-BST	HTR + $QO_{0.25}$	HTR + $QO_1 + M$
Abalone	# Nodes	**15.00 ± 2.00**	17.80 ± 2.28	18.60 ± 3.29	24.00 ± 5.96
	# Decision nodes	**6.00 ± 1.00**	7.40 ± 1.14	7.80 ± 1.64	6.00 ± 2.00
	# Leaves	**9.00 ± 1.00**	10.40 ± 1.14	10.80 ± 1.64	18.00 ± 4.06
	Height	4.80 ± 0.84	5.20 ± 0.45	5.40 ± 0.55	**3.60 ± 0.55**
Ailerons	# Nodes	**33.00 ± 8.37**	33.80 ± 11.88	34.60 ± 2.61	92.00 ± 20.55
	# Decision nodes	**16.00 ± 4.18**	16.40 ± 5.94	16.80 ± 1.30	22.00 ± 3.74
	# Leaves	**17.00 ± 4.18**	17.40 ± 5.94	17.80 ± 1.30	70.00 ± 17.29
	Height	**7.80 ± 0.84**	8.40 ± 1.95	7.80 ± 0.45	7.80 ± 0.84
Bike	# Nodes	74.20 ± 5.40	69.40 ± 16.82	65.00 ± 6.78	**33.20 ± 4.71**
	# Decision nodes	36.60 ± 2.70	34.20 ± 8.41	32.00 ± 3.39	**7.80 ± 1.30**
	# Leaves	37.60 ± 2.70	35.20 ± 8.41	33.00 ± 3.39	**25.40 ± 3.44**
	Height	11.60 ± 0.55	10.40 ± 0.89	10.00 ± 1.41	**4.20 ± 0.45**
CalHousing	# Nodes	**90.20 ± 9.86**	92.60 ± 17.05	93.40 ± 7.80	164.80 ± 18.51
	# Decision nodes	**44.60 ± 4.93**	45.80 ± 8.53	46.20 ± 3.90	45.00 ± 4.06
	# Leaves	**45.60 ± 4.93**	46.80 ± 8.53	47.20 ± 3.90	119.80 ± 15.22
	Height	10.00 ± 1.22	11.00 ± 1.00	9.80 ± 0.45	**7.80 ± 0.45**
Elevators	# Nodes	**62.20 ± 7.95**	66.60 ± 10.14	65.80 ± 4.60	88.40 ± 26.97
	# Decision nodes	30.60 ± 3.97	32.80 ± 5.07	32.40 ± 2.30	**18.80 ± 5.59**
	# Leaves	**31.60 ± 3.97**	33.80 ± 5.07	33.40 ± 2.30	69.60 ± 21.43
	Height	10.20 ± 1.30	10.80 ± 0.84	11.60 ± 0.55	**6.40 ± 1.14**
House8L	# Nodes	**125.00 ± 6.48**	133.00 ± 5.83	125.40 ± 11.95	217.00 ± 11.64
	# Decision nodes	**62.00 ± 3.24**	66.00 ± 2.92	62.20 ± 5.97	63.40 ± 3.78
	# Leaves	**63.00 ± 3.24**	67.00 ± 2.92	63.20 ± 5.97	153.60 ± 8.26
	Height	11.60 ± 0.55	10.60 ± 0.55	12.40 ± 1.67	**9.40 ± 0.89**
House16H	# Nodes	**118.60 ± 8.41**	121.80 ± 10.06	137.00 ± 6.16	211.60 ± 23.81
	# Decision nodes	58.80 ± 4.21	60.40 ± 5.03	68.00 ± 3.08	**56.40 ± 3.21**
	# Leaves	**59.80 ± 4.21**	61.40 ± 5.03	69.00 ± 3.08	155.20 ± 20.91
	Height	13.20 ± 1.30	15.20 ± 1.48	17.00 ± 1.00	**9.40 ± 0.89**
Metro	# Nodes	288.20 ± 18.54	287.80 ± 8.98	**249.00 ± 8.83**	262.60 ± 18.34
	# Decision nodes	124.80 ± 8.17	124.60 ± 3.58	106.40 ± 1.67	**93.40 ± 7.80**
	# Leaves	163.40 ± 11.97	163.20 ± 10.01	**142.60 ± 7.30**	169.20 ± 11.63
	Height	13.20 ± 1.64	12.80 ± 0.84	**11.80 ± 1.30**	12.20 ± 1.79
Pol	# Nodes	64.20 ± 6.42	**62.20 ± 5.02**	73.80 ± 7.01	74.80 ± 27.54

	# Decision nodes	31.60 ± 3.21	30.60 ± 2.51	36.40 ± 3.51	**15.40 ± 8.29**
	# Leaves	32.60 ± 3.21	**31.60 ± 2.51**	37.40 ± 3.51	59.40 ± 19.33
	Height	10.80 ± 0.45	10.80 ± 1.10	9.60 ± 0.55	**5.80 ± 1.30**
Wind	# Nodes	34.20 ± 2.28	**33.80 ± 3.35**	34.20 ± 1.10	35.00 ± 17.38
	# Decision nodes	6.60 ± 1.14	6.40 ± 1.67	6.60 ± 0.55	**5.40 ± 1.67**
	# Leaves	27.60 ± 1.14	**27.40 ± 1.67**	27.60 ± 0.55	29.60 ± 16.46
	Height	4.00 ± 0.00	4.00 ± 0.00	4.00 ± 0.00	**3.40 ± 0.55**
Wine	# Nodes	**28.20 ± 4.82**	29.40 ± 3.85	29.40 ± 4.34	38.60 ± 10.55
	# Decision nodes	13.60 ± 2.41	14.20 ± 1.92	14.20 ± 2.17	**9.80 ± 1.92**
	# Leaves	**14.60 ± 2.41**	15.20 ± 1.92	15.20 ± 2.17	28.80 ± 9.20
	Height	7.00 ± 1.22	6.80 ± 0.45	8.00 ± 0.71	**6.20 ± 0.84**
Friedman	# Nodes	523.80 ± 22.83	498.60 ± 7.27	**475.40 ± 21.65**	737.40 ± 66.12
	# Decision nodes	261.40 ± 11.41	248.80 ± 3.63	237.20 ± 10.83	**184.40 ± 16.43**
	# Leaves	262.40 ± 11.41	249.80 ± 3.63	**238.20 ± 10.83**	553.00 ± 49.70
	Height	16.80 ± 1.30	16.20 ± 0.84	22.20 ± 1.10	**6.40 ± 0.55**
MV	# Nodes	366.80 ± 18.63	354.00 ± 14.42	**253.60 ± 21.51**	261.60 ± 12.01
	# Decision nodes	182.40 ± 9.32	176.00 ± 7.21	125.80 ± 10.76	**94.00 ± 5.83**
	# Leaves	184.40 ± 9.32	178.00 ± 7.21	**127.80 ± 10.76**	167.60 ± 6.58
	Height	12.40 ± 0.55	13.00 ± 0.71	14.20 ± 0.84	**7.60 ± 0.55**
Puma8NH	# Nodes	51.80 ± 5.76	49.80 ± 2.28	48.60 ± 5.55	**29.00 ± 0.00**
	# Decision nodes	25.40 ± 2.88	24.40 ± 1.14	23.80 ± 2.77	**7.00 ± 0.00**
	# Leaves	26.40 ± 2.88	25.40 ± 1.14	24.80 ± 2.77	**22.00 ± 0.00**
	Height	9.40 ± 1.52	9.40 ± 0.89	10.20 ± 1.10	**4.00 ± 0.00**
Puma32H	# Nodes	43.40 ± 7.54	44.20 ± 8.32	38.60 ± 3.58	**22.60 ± 2.61**
	# Decision nodes	21.20 ± 3.77	21.60 ± 4.16	18.80 ± 1.79	**6.20 ± 1.79**
	# Leaves	22.20 ± 3.77	22.60 ± 4.16	19.80 ± 1.79	**16.40 ± 0.89**
	Height	8.40 ± 0.55	8.40 ± 0.89	8.40 ± 0.55	**3.40 ± 0.55**
Ranks	# Nodes				
	Avg.	**2.20**	2.40	2.27	3.13
	Synth.	3.50	3.00	**1.50**	2.00
	Real	**1.73**	2.18	2.55	3.55
	# Decision nodes				
	Avg.	2.60	3.00	2.93	**1.47**
	Synth.	3.75	3.25	2.00	**1.00**
	Real	2.18	2.91	3.27	**1.64**
	# Leaves				
	Avg.	**2.13**	2.33	2.27	3.27
	Synth.	3.50	3.00	**1.50**	2.00
	Real	**1.64**	2.09	2.55	3.73
	Height				
	Avg.	2.53	3.00	3.27	**1.20**
	Synth.	2.25	2.75	4.00	**1.00**
	Real	2.64	3.09	3.00	**1.27**

5.6.3 Airlines case study

E-BST did not finish the execution (around 10 million instances were processed) due to its elevated use of computational resources and time limitations. Nonetheless, most of the dataset was processed, enabling the reader to extrapolate the missing portions of the plot curves we will present.

We start our analysis by evaluating the performance metrics, as depicted in Fig. 16. In general, all HTRs had similar RMSE values, regardless of the chosen AO. The QO-based trees were much faster to train and spent less memory resources when compared to E-BST and TE-BST. The TE-BST-based HTR took almost twice the time of the QO-based HTRs to train. The E-BST-based HTR tree would probably be around three times slower than the QO-based trees. As previously observed in the benchmark datasets, TE-BST reduced the memory and time costs compared to E-BST. Once trained, the compared HTRs presented similar traverse times, regardless of the AO or the number of branches per split.

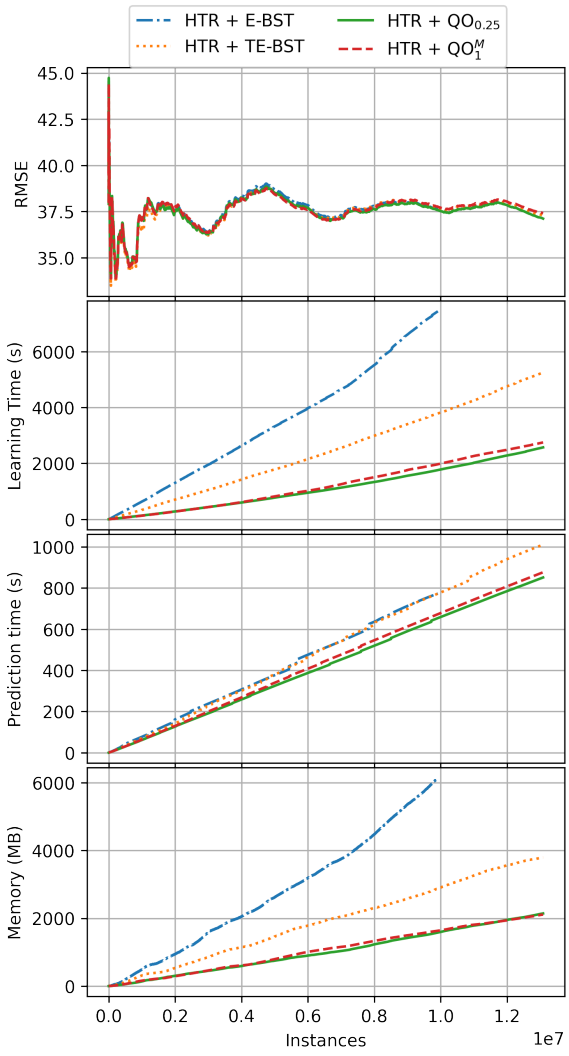


Figure 16 – Performance of the tree models in the airlines dataset.

Next, we inspect the obtained trees' characteristics in Fig. 17. Unsurprisingly, the QO_1^M-

based HTR was shallower than the E-BST and TE-BST ones. The $QO_{0.25}$-based HTR also did not reach the same heights as the E-BST variants, which is an unexpected observation. The split candidates produced by $QO_{0.25}$ did not offer significant gains in reducing the variance in y from some point in the processing onward. Nonetheless, as we did not observe an increase in RMSE, we argue that QO divided the input space more effectively when compared to E-BST and TE-BST. Hence, fewer split decisions were necessary to achieve similar predictive performance.

As expected, the number of nodes in the multi-branch tree was higher than the strictly binary ones. QO_1^M created more leaves than the other variants while keeping the number of decision nodes slightly inferior to the other trees. These observations are aligned with the results obtained in the previous sections.

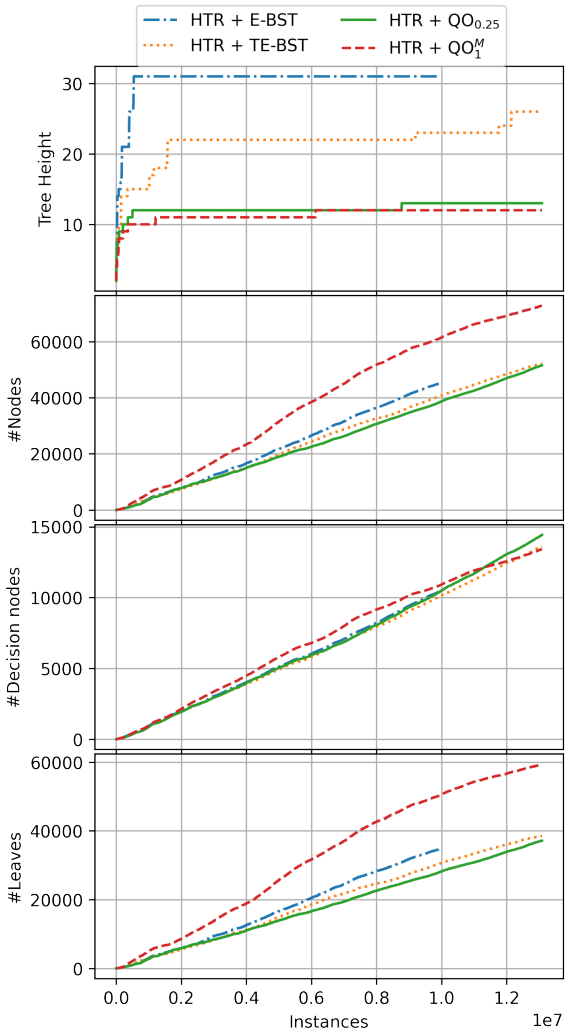


Figure 17 – Characteristics of the tree models in the airlines dataset.

Interestingly, we can see that the tree heights plateaued around 2 million processed instances. From this point onward, the QO-based trees had much more advantages regarding computational resource usage when compared to the E-BST variants. Indeed, QO has a sublinear memory cost, whereas E-BST (and to some extent TE-BST) has a linear cost on the number of

processed instances.

5.7 Conclusion

This paper evaluates the performance of Quantization Observer (QO) when building Hoeffding Tree Regressors (HTR). We also consider multi-branch HTRs built by using QO's partitions as tree branch prototypes. Two state-of-the-art Attribute Observer (AO) algorithms are compared against QO. Our experimental setup indicated that QO produces HTRs with a statistically equivalent prediction error while significantly decreasing memory usage and running time. The multi-branch enabled HTRs are shallower than the strictly binary ones, but their number of nodes is increased. Multi-branch trees deliver similar predictive performance to the strictly binary ones while further reducing computational resource usage. However, this observation depends on an adequate choice of the quantization radius. We evaluate multiple quantization radii and suggest values that provide a balance between predictive performance and computation resource usage.

As an open issue for future investigation, the QO's quantization radius is currently fixed in advance. Hence, all numerical features must be on the same scale. We intend to explore alternatives to adjust the radii during the tree-building process dynamically. We also intend to evaluate the QO usage in ensembles of HTRs to speed up training and reduce computational resource usage. Another appealing open issue for exploration is the impact of using multi-branch splits in Hoeffding Trees adapted to deal with concept drift. As we expect the multi-branch trees to be more compact than the strictly binary ones, concept drift adaptation routines could potentially impact predictive performance to lesser degrees. Finally, we intend to explore dynamic strategies to select split attempt intervals using multi-branch trees to save computational resources further.

Acknowledgments

This work was supported by São Paulo Research Foundation – FAPESP (grant #2018/07319-6).

CHAPTER

6

ONLINE EXTRA TREES REGRESSOR

Publication information: This chapter is an article published at IEEE Transactions on Neural Networks and Learning Systems journal.

Reference: MASTELINI, Saulo Martiello et al. Online Extra Trees Regressor. **IEEE Transactions on Neural Networks and Learning Systems**, © 2022 IEEE. Reprinted, with permission, from the authors.

6.1 Abstract

Data production has followed an increased growth in the last years, to the point that traditional or batch machine learning (ML) algorithms cannot cope with the sheer volume of generated data. Stream or online ML presents itself as a viable solution to deal with the dynamic nature of streaming data. Besides coping with the inherent challenges of streaming data, online ML solutions must be accurate, fast, and bear a reduced memory footprint. We propose a new decision tree-based ensemble algorithm for online machine learning regression named Online Extra Trees (OXT). Our proposal takes inspiration from the batch learning Extra Trees (XT) algorithm, a popular and faster alternative to Random Forest (RF). While speed and memory costs might not be a central concern in most batch applications, they become crucial in data stream data learning. Our proposal combines sub-bagging (sampling without replacement), random tree split points, and model trees to deliver competitive prediction errors and reduced

*<http://www.ieee.org/publications_standards/publications/rights/rights_link.html>

computational costs. Throughout an extensive experimental evaluation comprising 22 real-world and synthetic datasets, we compare OXT against the state-of-the-art adaptive RF (ARF) and other incremental regressors. OXT is generally more accurate than its competitors while running significantly faster than ARF and expending significantly less memory.

6.2 Introduction

With the increasing growth in data production, traditional or batch machine learning (ML) algorithms might be unfeasible for real-world applications due to their computational costs. Moreover, the generated data streams may change and evolve through time, i.e., the produced data is non-stationary (BAHRI *et al.*, 2021). This phenomenon, referred to as concept-drift (LU *et al.*, 2018) quickly renders the trained models outdated and inadequate to the task at hand. Online, incremental, or data stream-based ML algorithms aim to solve the mentioned challenges. These algorithms are fast, as each datum is only processed once, and they can deal with the data's non-stationary nature.

In the supervised and online ML setting, we assume data streams, DS, from which we sample instances in the form $(\mathbf{x}, y)_t$ at a given time or observation increment t. Each $\mathbf{x}$ consists of observations of a finite input feature set. Meanwhile, y represents one observation for the target value. In the case of continuous values for y, as in our case, the resulting learning task is called regression. Online ML algorithms seek to model a function h that approximates the unknown function $f : \mathbf{x} \mapsto y$. As we expect the data to be non-stationary, both f, as well as the generating distribution of $\mathbf{x}$ might change over time.

Regarding the incremental ML algorithms, decision trees are among the most popular ones in stream learning. More specifically, Hoeffding Trees (HT) are the most prominent family of incremental decision tree induction algorithms (DOMINGOS; HULTEN, 2000; IKONOMOVSKA; GAMA; DŽEROSKI, 2011b). HTs take this name as their tree induction algorithms rely on a statistical measure called Hoeffding bound to make split decisions. Tree-based ensembles are naturally popular extensions to boost standalone trees' prediction performance. In fact, multiple HT-based ensembles were proposed throughout the years (GOMES *et al.*, 2017; KRAWCZYK *et al.*, 2017; GOMES *et al.*, 2021). Most of the existing solutions target classification tasks, but a few concern regression tasks (GOMES *et al.*, 2018; GOMES *et al.*, 2020), which is our focus in this paper.

Ensembles are known to work well, provided that the individual learners are accurate and diverse (SAGI; ROKACH, 2018). Unfortunately, we cannot treat randomization and diversity induction the same as in batch learning applications. As showed by Gomes et al. (GOMES *et al.*, 2021), the ubiquitous HTs are stable learners. Thus, we cannot assume that the same strategies used by ensembles of Classification and Regression Trees (CART) (BREIMAN *et al.*, 2017; BREIMAN, 1996), the standard base learner choice for batch tree ensembles, would

also straightforwardly work for ensembles of HTs. Creating efficient and effective tree-based ensembles for online learning remains an open issue. The main focus to induce diversity among the base learners lies in modifying HTs to add some source of randomization. This work goes one step further towards this objective compared to existing tree-based ensembles.

Many of the popular tree ensemble strategies for batch ML were already adapted to online scenarios. They include Bagging (OZA; RUSSELL, 2001b; BIFET; HOLMES; PFAHRINGER, 2010), Random Forests (GOMES *et al.*, 2017; GOMES *et al.*, 2018), Random Patches (GOMES *et al.*, 2020; GOMES *et al.*, 2021), among others. Extra Trees (XT) (GEURTS; ERNST; WEHENKEL, 2006) were not to this date adapted for data stream applications. Due to its random nature, XT is often associated with a reduced memory footprint and training time, whereas its predictive performance is still competitive or even superior. These characteristics are often overlooked in batch applications. Nonetheless, they are especially appealing for online ML.

XTs explore random split points when building their base learners. While this is a simple idea, applying the same strategy in stream learning is not trivial. The data range is unknown beforehand, and it may change as more data arrives. Therefore, an approximation procedure must be devised to infer split candidates within the data range. We also explore an instance sampling alternative to online bagging to reduce computational costs further.

To create random split decisions, as done in the original XT algorithm, we need to change the splitting strategy utilized by the HT regressors. HTs rely on Attribute Observers (AO) to enable online splits and allocate an instance of AO per feature in each of their leaves. AOs are responsible for defining split candidates and maintaining target statistics for them. HT regressors rely on the Variance Reduction heuristic to perform splits (IKONOMOVSKA; GAMA; DŽEROSKI, 2011b; MASTELINI; CARVALHO, 2021). Thus they keep incremental estimates of the target variance for each split candidate. The current best split candidates for each feature are compared during split attempts. The best split point and feature combination are selected for expanding the tree structure. In this work, we propose a new AO to keep statistics and perform random splits in OXT. The proposed AO keeps a small buffer of instances used to estimate the range of the features and define random split points.

Our main contributions in this work can be summarized as follows:

1. A new algorithm for online regression learning that provides experimental results either superior or similar to state-of-art related regressors;

2. The first adaptation of Extra-Trees to online learning. Specifically, it combines incrementally defined random splits and sub-bagging to enhance the predictive performance and decrease computational costs;

3. A comparison using 22 datasets between our proposed algorithm, a state-of-the-art tree-based incremental ensemble algorithm, and other incremental regressors to assess the

effectiveness of our proposal using an extensive experimental setup;

4. A publicly available implementation of our proposed algorithm [2].

This work is organized as follows. Section 6.3 introduces background and research proposals related to our new tree-based online ensemble algorithm. We describe our proposal, OXT, in section 6.4 and our experimental setup to assessing its performance in section 6.5. The obtained results are presented and discussed in section 6.6. Finally, we present our concluding remarks and future works in section 6.7.

6.3 Background

Ensemble learning is a popular topic in both batch ML (SAGI; ROKACH, 2018; DONG *et al.*, 2020) and Online ML (KRAWCZYK *et al.*, 2017; GOMES *et al.*, 2020; GOMES *et al.*, 2021). By combining potentially inaccurate base learners, ensembles can reach high predictive scores, however, at the cost of increased computational cost. Hence, reducing online ensembles' memory footprint and running time is an important research focus. Online ensemble algorithms are usually more resilient against concept drift, noisy data, and outliers (KRAWCZYK *et al.*, 2017; DONG *et al.*, 2020) than standalone predictors. These characteristics make ensembles even more appealing for dynamic streaming applications.

The literature presents multiple strategies to combine base models. The first distinction we need to make is about the choice of the base learners. Ensembles can be heterogeneous or homogeneous regarding their base learners (GOMES *et al.*, 2017). The former option combines different types of base learners leveraging their individual advantages. It is in charge of the user to pick a good combination and number of base learners, including their configuration. The reader may refer to the surveys (KRAWCZYK *et al.*, 2017), and (GOMES *et al.*, 2017) for more information about this kind of online ensemble solution.

The second group of ensemble algorithms combine base learners of the same type and apply data perturbations to induce diversity among its constituent models. It is usual to select unstable learners in this group because their predictions can be highly affected by small changes in the incoming instances. The two most prominent families of homogeneous ensembles are bagging and boosting (BREIMAN, 1996; FRIEDMAN, 2001; OZA; RUSSELL, 2001b).

Due to the sequential dependency of its base models, boosting was not thoroughly explored in data stream learning scenarios (OZA; RUSSELL, 2001b; GOMES *et al.*, 2017). Online bagging, on the other hand, is one of the most popular choices for online ensemble learning (BIFET; HOLMES; PFAHRINGER, 2010; GOMES *et al.*, 2017; GOMES *et al.*, 2021). Although our proposal uses homogeneous base learners, it does not rely on an online bagging

[2] OXT is available in River.

strategy, as we discuss in section 6.4. Nonetheless, the trees in OXT also perform instance sampling, so our proposal is closely related to bagging-based ensembles. Next, we review the main bagging and tree-based online ensembles algorithms related to our proposal.

6.3.1 Building ensembles via instance re-sampling

The seminal work by Oza and Russel (OZA; RUSSELL, 2001b) introduced the notion of online bagging. The authors noted that the sampling effect of batch bagging could be approximated online by sampling instance weights from a Poisson distribution parameterized with $\lambda = 1$. These weights represent the number of times a given base learner will process each instance in the ensemble. Some weights might equal zero. Thus, the model will ignore these instances, as in traditional bagging.

Leveraging Bagging (LB) extended online bagging by increasing the λ parameter of the Poisson distribution to 6 (BIFET; HOLMES; PFAHRINGER, 2010). This change results in an increased re-sampling rate that potentially increases the predictive performance, as base models process each datum multiple times. The increase in predictive performance comes at the cost of increased computational costs. LB also introduces a concept of output codes to decompose classification tasks into binary strings and train binary classifiers for each portion of the string code. However, this aspect has no direct application in regression tasks.

As a promising alternative to bagging, sub-bagging (ZAMAN; HIROSE, 2009; YATES; ISLAM, 2021) presents itself as a lightweight and faster model diversity induction strategy for batch-based ensembles. To the best of our knowledge, sub-bagging has not yet been evaluated in online ML scenarios. Different from bagging, when using sub-bagging, instances are sampled without replacement. At first glance, this difference may appear minimal. Notwithstanding, we hypothesize that by combining this sampling strategy with the learning stability inherent to HTs, the tree regressors can process fewer data without losing predictive performance.

6.3.2 Tree-based online ensembles algorithms

Random Forests (RF) are among the most popular bagging and tree-based ensemble algorithms for batch learning. Naturally, multiple realizations of this algorithm for data stream learning were proposed in the last years. For instance, the reader may refer to (WANG *et al.*, 2009), (MOURTADA; GAÏFFAS; SCORNET, 2021), as well as other exemplars mentioned in (GOMES *et al.*, 2017). Gomes et al. (GOMES *et al.*, 2017) introduces Adaptive Random Forest (ARF), which combines the strengths of LB with the RF algorithm. A version of ARF for regression was also proposed (GOMES *et al.*, 2018). ARF is to date the most popular online RF algorithm, being even used as a building block of incremental Deep Forests (LUONG; NGUYEN; LIEW, 2020; KORYCKI; KRAWCZYK, 2020).

Another popular bagging-based incremental ensemble is Streaming Random Patches

(SRP) (GOMES *et al.*, 2020; GOMES *et al.*, 2021). Sharing many similarities with ARF, SRP can use arbitrary base learners rather than only decision trees. Moreover, SRP applies a different type of data perturbation to induce its base models. While ARF (and RF) sample random subsets of input features to be evaluated as split candidates at each tree node, SRP (and its batch version, Random Patches) selects a single random feature subset to build the entire base model. When limiting the analysis to decision trees, one may argue that ARF can induce more diversity among its base learners compared to SRP. However, SRP has the flexibility of applying any kind of base learner.

Our proposal, OXT, combines random tree splits, similar to those used by ARF, and sub-bagging sampling to obtain a lightweight, fast, and highly accurate ensemble regressor.

6.4 Online Extra Trees

We will now introduce the characteristics of our proposal, highlighting its differences from current tree-based ensembles. We follow a top-down explanation where we introduce first the ensemble as a whole, followed by the general characteristics of the base learners, and lastly, the split criterion.

6.4.1 Ensemble characteristics

Inspired by the original XT and also by ARF, we present the training procedure of OXTs in Algorithm 3.

Firstly, OXT relies on sub-bagging (ZAMAN; HIROSE, 2009; YATES; ISLAM, 2021) rather than online bagging, as employed by ARF, to build its trees (line 17, Algorithm 3). Online bagging approximates the traditional batch-based bagging approach by sampling the instance weights from Poisson distributions (OZA; RUSSELL, 2001b). In sub-bagging, instances are sampled without replacement. Therefore, the trees will only process a subset of the instances similarly to bagging. However, the selected instances are only processed once rather than multiple times in this case. In Algorithm 3, `rand` represents a procedure that samples uniformly between $[0, 1]$. The approximate percent of instances each tree will observe is controlled by the user as a hyperparameter, p.

Another crucial step is the selection of appropriate drift detection and countermeasure strategies. We follow the trend initiated in ARF (GOMES *et al.*, 2017) and then replicated to SRP (GOMES *et al.*, 2021) and others ensembles to deal with concept drifts. OXT keeps two levels of detectors for each tree: warning and drift. The former is more sensitive to changes than the latter one. These differences can usually be achieved by setting different concept drift confidence levels in the selected detectors.

When a warning is triggered, a new tree begins to be trained in the background (line 11,

Algorithm 3 – Online Extra Trees training.

Require: N_t: number of trees, p: chance of selecting instance in sub-bagging, DS: data stream containing input features (x) and target values (y)

1: **function** OXT-TRAIN(N_t, p, DS)
2: $T \leftarrow \emptyset$ ▷ Empty forest
3: **for** $t \leftarrow 1, N_t$ **do**
4: Initialize t-th empty HT and add it to T
5: **end for**
6: **while** DS has instances **do**
7: $(x, y) \leftarrow$ `next`(DS)
8: **for** $t \leftarrow 0, N_t$ **do**
9: $\hat{y} \leftarrow$ `pred`(T_t, x)
10: $warn, drift \leftarrow$ `detect_change`(y, $\hat{y}$)
11: **if** $warn$ **then**
12: Build new tree, T_t^b, in the background
13: **end if**
14: **if** $drift$ **then**
15: Delete outdated tree and replace it by the background tree
16: **end if**
17: **if** `rand()` $> p$ **then** ▷ Skip training with this instance for this tree
18: `continue`
19: **end if**
20: rHT-Train(T_t, x, y) ▷ Train modified HT with the instance
21: **if** T_t^b exists **then**
22: rHT-Train(T_t^b, x, y)
23: **end if**
24: **end for**
25: **end while**
26: **return** T
27: **end function**

Algorithm 3). If the warning escalates to drift, the tree is discarded and replaced by the new one (line 14). The user can select the appropriate warning and drift detectors. Although one could rely on (randomized) periodic drift detectors, as reported by Gomes et al. (GOMES *et al.*, 2020), and, thus, potentially add another level of randomness, we decided to use the same state-of-the-art active drift detection strategy adopted by ARF and SRP. We did not investigate other concept drift adaptation strategies because our primary goal with OXT is to decrease memory usage and running time while maintaining predictive accuracy. Nonetheless, one can easily change the drift detectors used in OXT. In OXT, the drift detectors monitor the absolute deviations between the true y values and the predictions.

After performing concept drift detection, reacting to it, and sampling instances via sub-bagging, the next step consists of updating the base learners per se. This is represented by the procedure `rHT-Train`, i.e., random HT training. `rHT-Train` updates a base tree with a single instance. Both the main base learners (line 20) and the background trees (line 22) rely on this

procedure.

ARF and OXT share many similarities in their tree induction strategies. However, the split decisions in OXT's trees are random, whereas, in ARF, they are deterministic. We detail the main aspects of OXT's tree induction, as well as the differences to ARF, in the next subsections.

The final predictions of OXT are obtained by averaging the predictions of individual trees.

6.4.2 Hoeffding Tree modifications

OXT also modifies the original HT regressor to induce randomness. The first modification consists in evaluating only a subset of features at each split attempt. Each leaf node only considers a random subset of features (sampled without replacement) when attempting a split. ARF modifies its base learners in the same way.

In online ML algorithms, the benefits of using subsets of features are twofold. Firstly, this action should induce diversity among the base learners as each tree will be built on different data views. Secondly, this action usually decreases the computational costs of the trees, as AOs are only kept for the selected subsets of the features in the leaves. As discussed in (MASTELINI; CARVALHO, 2021), AOs are one of the primary resource-intensive portions of the HTs.

Aside from the mentioned aspects, the remaining elements of OXT's HTs are the same as the vanilla HT regressor. For more details on HT regressor construction, the reader is referred to Ikonomovska et al. (IKONOMOVSKA; GAMA; DŽEROSKI, 2011b) and Mastelini and Carvalho (MASTELINI; CARVALHO, 2021).

The next main modification OXT applies to its base learners concerns how the split decisions are made. The procedure `rHT-Train` (lines 20 and 22, Algorithm 3) works by performing the HT induction algorithm while using subsets of features, as aforementioned, and relying on the random split-based AO described next.

6.4.3 Performing random splits online

HTs, ARF, and other ensembles of HTs for regression rely on deterministic strategies to perform split decisions. An exhaustive enumeration-based AO was initially used to build HT regressors (IKONOMOVSKA; GAMA; DŽEROSKI, 2011b). In this approach, every observed input and target values are stored in a binary search tree (BST). This AO is dubbed Extended BST (E-BST), as it relies on BSTs and stores target-related information. Tree traversals retrieve each entry's split merit, and the best candidate is selected for splitting. More efficient strategies were proposed to alleviate the costs of E-BST by Mastelini and Carvalho (MASTELINI; CARVALHO, 2021). These strategies use either BSTs or quantization strategies to speed up HT regressor building. OXT, inspired by the original XT algorithm, relies instead on randomly defined splits.

It is trivial to create random uniform splits in a batch setting. One only needs to know the data range beforehand, and this can be done in a single pass over the dataset. However, in an online setting, data is constantly changing, and the observed range might differ from time to time. Besides, as tree splits are performed, the regions of the input space become increasingly narrow.

As a solution, we propose a new AO that applies a semi-lazy approach to obtain estimates of the feature range. We refer to this AO as *Random Splitter*. Our procedure to update the Random Splitter is presented in Algorithm 4, and it is called by `rHT-Train`. When a new split is performed and new leaves created, each newly instantiated Random Splitter stores a small buffer with the first arriving observations, as shown in line 9. The size of such buffer, s, is a user-given parameter. Once the buffer is filled up, the stored elements are used to estimate the range of the input data (line 12).

Algorithm 4 – Random Splitter update function.

Require: s: buffer size, DS_f: a stream consisting of observations from a single feature x_f and the target y

```
1:  function RANDOM-SPLITTER-UPDATE(s, DS_f)
2:      B ← ∅                                              ▷ Empty buffer
3:      init ← False
4:      v_≤, v_> ← var(), var()          ▷ Initialize the incremental variance estimators
5:      while DS_f has instances do
6:          (x_f, y) ← next(DS_f)
7:          if not init then
8:              if |B| < s then
9:                  Add (x_f, y) to B
10:                 continue
11:             else
12:                 Set a threshold φ by sampling uniformly from the range of x_f in B
13:                 Replay elements of B updating v_≤ and v_> accordingly to φ
14:                 Delete B
15:                 init ← True
16:             end if
17:         end if
18:         Update v_≤ with y if x_f ≤ φ else use y to update v_>
19:     end while
20:     return φ, v_≤, v_>
21: end function
```

With the approximate range, we can choose a random split point (ϕ) uniformly in the given range. From this point onward, we do not need to store the inputs but update the splitting enabling statistics for elements smaller than or equal to ($\leq$) or greater than ($>$) the ϕ value (line 18). The AO replays the elements stored in the buffer to account for them in the split statistics (line 13).

In Algorithm 4, `var` represents the incremental variance estimators built upon the renowned Welford's algorithm, as presented in Mastelini and Carvalho (MASTELINI; CAR-

VALHO, 2021). These estimators can be manipulated to calculate the Variance Reduction heuristic, which is used to estimate the merit of each split candidate. By using Random Splitter as its AO, the randomized HT regressors only consider one split candidate per monitored feature, ϕ, at the tree leaves.

6.5 Experimental setup

This section describes the experimental setup applied in our empirical comparison of OXT against its contenders. We detail the used datasets, performance metrics, baselines, and comparison strategies.

The OXT code was written in Python, and we relied on the River library to perform our experiments (MONTIEL *et al.*, 2021). Our experiments were performed on a machine running CentOS with two Intel Xeon E5-2667v4 processors with 8 cores and running at 3.2 GHz. The machine has 512 Gigabytes of DDR3 (1866MHz) memory. All algorithms ran sequentially in our comparisons.

6.5.1 Datasets

In our study, we explored synthetic and real-world datasets commonly used in the data stream regression literature (IKONOMOVSKA; GAMA; DŽEROSKI, 2011b; GOMES *et al.*, 2018; GOMES *et al.*, 2020). In total, we considered 10 real-world stationary datasets and 12 synthetic datasets, from which 8 are non-stationary. The synthetic datasets Hyper(A), Hyper(G), Hyper(I), RBG(A), RBF(G) were previously used in Gomes et al. (GOMES *et al.*, 2020) and contain abrupt, incremental, and gradual concept drifts. The non-stationary Friedman variants comprise abrupt and gradual concept drifts. These synthetic datasets were proposed by Ikonomovska et al. (IKONOMOVSKA; GAMA; DŽEROSKI, 2011b). The data generators are available in River. LEA has three concept drifts (after 25k, 50k, and 75k instances in our case), while the GRA and GSG variants define two concept drift points (after 35k and 75k instances, in our case). In the case of the dataset with gradual concept drifts, a transition window of 2000 instances between concepts was utilized.

More details about the considered datasets are summarized in Table 17.

The categorical features were one-hot-encoded in all cases, and we also applied an incremental version of standard scaling. In other words, all features were firstly centered by a running estimate of their mean value and scaled by an incremental estimate of the standard deviation.

Table 17 – Characteristics of the evaluated datasets. Missing mean and standard deviation of the target implies generator-based datasets. LEA: Local-expanding Abrupt; GRA: Global Re-occurring Abrupt; GSG: Global and Slow Gradual.

Dataset	#Instances	#Numeric features	#Categorical features	Mean ± SD.
Real-world datasets				
Abalone	4977	7	1	10.067 ± 3.325
Ailerons	13750	40	0	-0.001 ± 0.00
Bike	17379	12	0	189.463 ± 181.388
CalHousing	20500	8	0	$207502.124 \pm 115206.782$
Elevators	16599	18	0	0.022 ± 0.007
House8L	22784	8	0	50074.440 ± 52843.476
House16H	22784	16	0	50074.440 ± 52843.476
Metro	48204	4	3	3259.818 ± 1986.861
Wind	6574	12	2	15.600 ± 6.698
Wine	5298	11	0	5.871 ± 0.891
Synthetic datasets				
Pol	15600	48	0	29.065 ± 41.795
Puma8NH	8192	8	0	1.162 ± 5.622
Puma32H	8192	32	0	0.00 ± 0.030
Friedman	100000	10	0	–
Friedman(LEA)	100000	10	0	–
Friedman(GRA)	100000	10	0	–
Friedman(GSG)	100000	10	0	–
Hyper(A)	500000	10	0	31.175 ± 7.483
Hyper(G)	500000	10	0	31.176 ± 7.483
Hyper(I)	500000	10	0	-63.574 ± 93.821
RBF(A)	500000	20	0	50.202 ± 28.266
RBF(G)	500000	20	0	50.199 ± 28.271

6.5.2 Evaluation strategy

We have used the prequential evaluation approach throughout our experiments. We performed five repeated runs of the algorithms and datasets in all the cases. This includes our general comparisons and other measurements and studies performed to extract information from the trained ensembles. In the case of the stationary datasets, each dataset was pseudo-shuffled at each run using reservoir sampling with a window of 100 instances. When evaluating non-stationary datasets, no data shuffling was applied, although the generator-based datasets were synthesized using different random seeds for each run.

We measured the Root Mean Square Error (RMSE), the Coefficient of Determination (R^2), the memory usage, and the running time of the compared algorithms. We reported the results obtained by averaging the measurements of the repeated executions.

No label delay (GOMES *et al.*, 2022) was considered in our setup. Although the evaluation of such scenarios is crucial, it is out of the scope of this work.

6.5.3 Contenders and baselines

We detail next the online regressors compared in our experimental setup:

- **O**nline E**X**tra **T**rees: Our proposed algorithm, referred to simply as OXT;
- **A**daptive **R**andom **F**orest: An online tree-based ensemble inspired on the traditional Random Forest. This algorithm is considered as the current state-of-art (GOMES *et al.*, 2017; GOMES *et al.*, 2018), and here is referred to as ARF;

- **A**daptive **M**odel **Rules**: An algorithm that employs the Hoeffding bound and variance-ratio to build rule sets. Each rule is complemented with an adaptive linear model to perform predictions (DUARTE; GAMA; BIFET, 2016). Here it is referred to as AMRules;

- **H**oeffding **T**ree regressor: A baseline which consists of an standalone HT (IKONOMOVSKA; GAMA; DŽEROSKI, 2011b; GOMES *et al.*, 2020). For simplicity, this algorithm is referred to as HT;

- **P**assive **A**ggressive **R**egressor: A well-renowned incremental linear model referred to as PAR (CRAMMER *et al.*, 2006);

- Dummy: A simple regressor that always produces an incremental estimate of the mean target value.

Gomes et al. (GOMES *et al.*, 2020) compare ARF against SRP and other bagging-based ensemble variants on regression tasks. The authors report that SRP was not better than ensembles of bagging-based HT regressors. ARF, however, was one of the most accurate algorithms among those compared. We only consider ARF as the main tree-based ensemble contender to OXT in our work. Our claim bears from both the results reported in (GOMES *et al.*, 2020) and the similarities between RF and the batch XT algorithm (GEURTS; ERNST; WEHENKEL, 2006). Furthermore, we have also considered the prominent rule-based method AMRules, and the HT regressor as a standalone tree-based baseline comparison.

ARF, AMRules, and HT relied on the Truncated Extended-Binary Search Tree (TE-BST) (MASTELINI; CARVALHO, 2021) as their AO algorithm. TE-BST works as a wrapper over the traditional E-BST AO (IKONOMOVSKA; GAMA; DŽEROSKI, 2011b; MASTELINI; CARVALHO, 2021) and aims at speeding up the latter algorithm. To do so, TE-BST firstly rounds the inputs to a user-given number of decimal places and then passes the transformed input to E-BST. E-BST stores the inputs and variance estimates of the target in a BST structure that enables evaluating split candidates. We set TE-BST to round the inputs to one decimal place.

As the drift detector for OXT, AMRules and, ARF, we have employed ADWIN (BIFET; GAVALDA, 2007). For ARF and OXT, we set the confidence levels to $\delta = 0.01$ and $\delta = 0.001$ for the warning and drift detectors, respectively. AMRules used the default confidence value ($\delta = 0.002$), as implemented in River.

Both HT, AMRules, ARF, and OXT use adaptive predictors (target mean or a linear model). Faded error metrics are used to weigh the influence of past and new observations when selecting the best predictor. A fading factor of 0.95 was selected, as previously reported in related literature (OSOJNIK; PANOV; DŽEROSKI, 2018).

When not explicitly stated otherwise, all the compared algorithms' hyperparameters were kept fixed to their default values, as defined in River (MONTIEL *et al.*, 2021).

6.5.4 Ensemble-specific configurations

OXT and ARF used 20 trees for the main experimental evaluation. Their trees adopted the same set of hyperparameters used in the standalone HT regressor. An exception was the split attempt interval (grace period) set to 50 in ARF and OXT.

Due to the random nature of the splits performed by OXT, the split merits ought to be less promising than those obtained from a deterministic strategy, such as the one used by AMRules, ARF, and HT. For this reason, we set the split significance value, δ, of OXT's trees to 0.05. ARF uses $\delta = 0.01$, whereas HT and AMRules use $\delta = 10^{-7}$ by default. Moreover, we disabled the pre-pruning capabilities of the trees in OXT. HTs, including the ones in ARF, usually consider the option of not expanding the tree structure during split attempts. Specifically, HTs consider that the best split candidate should significantly reduce the current target variance accordingly to the Hoeffding bound. If that is not the case, the tree structure is not changed. We disabled this test in OXT as we want to deploy multiple random splits of the input space to increase model diversity.

The remaining hyper-parameters used in the HTs were kept at their standard values (as implemented in River), and the same values were applied to both ARF and OXT. Both ARF and OXT use random feature subsets when performing split attempts. The size of these subsets was set to $\sqrt{M}$, where M represents the number of input features.

Also, ARF aggregated the predictions of individual trees using the median operation. As reported in the empirical studies of Gomes et al. (GOMES *et al.*, 2018), the median aggregation might favor model tree-based ensembles. Despite that, we aimed for simplicity and computational efficiency. Hence, OXT used the straightforward mean aggregation when joining the predictions of individual trees.

Each tree in OXT used sub-bagging with 50% of the input data (parameter p in Algorithm 3), as suggested by Zaman and Hirose (ZAMAN; HIROSE, 2009). The ARF re-sampling strategy was the same used in Leveraging Bagging (BIFET; HOLMES; PFAHRINGER, 2010) (Poisson sampling with $\lambda = 6$). Concerning OXT, the size of the buffer used to perform the splits was set to 5 (parameter s in Algorithm 4).

6.6 Results and discussion

Initially, we present a comparison between our proposed method and its contenders. Following, we extend our analysis by performing a saturation study between our method and ARF. Likewise, we also compare the characteristics of the forests generated by ARF and OXT.

6.6.1 General performance comparison

As can be seen in Tables 18 and 19, our proposed method, OXT, is often associated with the best performance. More specifically, OXT outperforms its main competitor, ARF, in 14 out of 22 datasets considering RMSE, whereas ARF yields superior performance only in 4 cases. Our results are further highlighted in the Friedman-Nemenyi test presented in Figure 18 where our method is ranked in the first position. The R^2 results also support the edge OXT has over ARF concerning predictive performance, and highlight the goodness of the regression fit our proposal can deliver in most cases.

Table 18 – RMSE results

Dataset	PAR	Dummy	HT	AMRules	ARF	OXT
Abalone	5.560 ± 0.03	3.341 ± 0.00	**2.314 ± 0.02**	2.362 ± 0.05	2.363 ± 0.02	2.573 ± 0.03
Ailerons	1.007 ± 0.01	**0.000 ± 0.00**	**0.000 ± 0.00**	**0.000 ± 0.00**	**0.000 ± 0.00**	**0.000 ± 0.00**
Bike	140.460 ± 0.32	181.597 ± 0.02	130.963 ± 14.08	336.055 ± 235.69	112.434 ± 1.18	**110.103 ± 3.61**
CalHousing	226188.356 ± 15.86	115229.217 ± 2.94	71177.598 ± 1705.69	81330.050 ± 2600.65	**63071.461 ± 376.80**	65865.563 ± 116.18
Elevators	0.9308 ± 0.01	0.007 ± 0.00	0.005 ± 0.00	0.006 ± 0.00	0.005 ± 0.00	**0.004 ± 0.00**
House8L	65898.359 ± 32.82	52868.176 ± 37.97	39948.989 ± 956.62	57164.943 ± 4163.46	37002.294 ± 163.49	**35178.489 ± 122.11**
House16H	66064.4393 ± 32.26	52868.176 ± 37.97	43745.180 ± 264.19	51024.655 ± 1004.62	**40819.751 ± 94.99**	41882.176 ± 304.90
Metro	1999.706 ± 4.39	1987.202 ± 0.06	2134.410 ± 60.65	1984.154 ± 1.67	4546.153 ± 408.57	**1930.133 ± 8.34**
Pol	39.191 ± 0.10	41.812 ± 0.01	**24.239 ± 0.45**	42.618 ± 1.03	28.197 ± 0.76	24.898 ± 1.08
Wind	9.661 ± 0.04	6.690 ± 0.01	5.079 ± 0.16	6.511 ± 0.11	5.901 ± 3.08	**4.326 ± 0.12**
Wine	4.481 ± 0.06	0.896 ± 0.00	0.807 ± 0.01	0.830 ± 0.01	**0.740 ± 0.01**	0.769 ± 0.00
Friedman	7.524 ± 0.01	4.983 ± 0.01	**1.929 ± 0.05**	2.590 ± 0.03	2.333 ± 0.02	2.0770 ± 0.02
Puma8NH	7.651 ± 0.05	5.627 ± 0.00	3.839 ± 0.08	4.329 ± 0.05	3.848 ± 0.06	**3.773 ± 0.03**
Puma32H	0.789 ± 0.00	0.030 ± 0.00	0.018 ± 0.00	0.026 ± 0.00	0.029 ± 0.00	**0.0176 ± 0.00**
Friedman(LEA)	7.815 ± 0.00	5.446 ± 0.00	2.327 ± 0.00	2.915 ± 0.00	2.721 ± 0.03	**2.319 ± 0.02**
Friedman(GRA)	7.541 ± 0.00	4.981 ± 0.00	2.274 ± 0.00	2.690 ± 0.00	2.465 ± 0.01	**2.249 ± 0.01**
Friedman(GSG)	7.552 ± 0.00	4.995 ± 0.00	**2.150 ± 0.00**	2.657 ± 0.00	2.506 ± 0.02	2.275 ± 0.01
Hyper(A)	7.343 ± 0.00	7.483 ± 0.00	2.289 ± 0.00	2.278 ± 0.00	2.442 ± 0.05	**1.879 ± 0.01**
Hyper(G)	7.553 ± 0.00	7.484 ± 0.00	2.548 ± 0.00	2.650 ± 0.00	2.896 ± 0.04	**2.238 ± 0.01**
Hyper(I)	**44.955 ± 0.00**	93.821 ± 0.00	48.785 ± 0.00	47.393 ± 0.14	45.154 ± 0.06	49.655 ± 0.09
RBF(A)	33.902 ± 0.00	28.267 ± 0.00	**15.213 ± 0.00**	25.299 ± 0.05	20.344 ± 0.05	21.743 ± 0.05
RBF(G)	34.029 ± 0.00	28.271 ± 0.00	**15.443 ± 0.00**	25.844 ± 0.00	20.500 ± 0.07	21.858 ± 0.08
Avg. rank	5.45	4.91	2.32	3.86	2.68	**1.77**
Avg. rank real	5.45	4.64	2.73	4.18	2.36	**1.64**
Avg. rank synth.	5.45	5.18	**1.91**	3.55	3.00	**1.91**

Table 19 – R2 results

Dataset	PAR	Dummy	HT	AMRules	ARF	OXT
Abalone	−1.779 ± 0.03	−0.005 ± 0.00	**0.518 ± 0.01**	0.498 ± 0.02	0.497 ± 0.01	0.404 ± 0.01
Ailerons	−6086104.578 ± 145810.78	−0.001 ± 0.00	0.467 ± 0.02	−0.293 ± 0.88	0.599 ± 0.01	**0.650 ± 0.10**
Bike	0.402 ± 0.00	−0.000 ± 0.00	0.475 ± 0.12	−3.772 ± 6.41	0.617 ± 0.01	**0.632 ± 0.02**
CalHousing	−2.855 ± 0.00	−0.000 ± 0.00	0.618 ± 0.02	0.501 ± 0.03	**0.700 ± 0.00**	0.673 ± 0.00
Elevators	−19191.089 ± 309.93	−0.001 ± 0.00	0.371 ± 0.47	0.218 ± 0.50	0.544 ± 0.00	**0.658 ± 0.13**
House8L	−0.554 ± 0.00	−0.000 ± 0.00	0.429 ± 0.03	−0.175 ± 0.17	0.510 ± 0.00	**0.557 ± 0.00**
House16H	−0.562 ± 0.00	−0.000 ± 0.00	0.3150 ± 0.01	0.068 ± 0.04	**0.404 ± 0.00**	0.372 ± 0.01
Metro	−0.013 ± 0.00	−0.000 ± 0.00	−0.155 ± 0.07	0.003 ± 0.00	−4.269 ± 0.97	**0.056 ± 0.01**
Pol	0.121 ± 0.00	−0.001 ± 0.00	**0.664 ± 0.01**	−0.040 ± 0.05	0.545 ± 0.02	0.645 ± 0.03
Wind	−1.090 ± 0.02	−0.002 ± 0.00	0.422 ± 0.04	0.050 ± 0.03	0.051 ± 0.95	**0.581 ± 0.02**
Wine	−24.287 ± 0.60	−0.010 ± 0.00	0.179 ± 0.01	0.133 ± 0.02	**0.3100 ± 0.01**	0.255 ± 0.00
Friedman	−1.281 ± 0.01	−0.000 ± 0.00	**0.850 ± 0.01**	0.730 ± 0.01	0.781 ± 0.00	0.826 ± 0.00
Puma8NH	−0.851 ± 0.02	−0.002 ± 0.00	0.534 ± 0.02	0.407 ± 0.01	0.532 ± 0.01	**0.550 ± 0.01**
Puma32H	−679.301 ± 1.72	−0.001 ± 0.00	0.631 ± 0.02	0.253 ± 0.11	0.097 ± 0.02	**0.663 ± 0.02**
Friedman(LEA)	−1.060 ± 0.00	−0.00 ± 0.00	0.817 ± 0.00	0.714 ± 0.00	0.750 ± 0.01	**0.819 ± 0.00**
Friedman(GRA)	−1.293 ± 0.00	−0.000 ± 0.00	0.792 ± 0.00	0.708 ± 0.00	0.755 ± 0.00	**0.796 ± 0.00**
Friedman(GSG)	−1.286 ± 0.00	−0.000 ± 0.00	**0.815 ± 0.00**	0.717 ± 0.00	0.748 ± 0.00	0.792 ± 0.00
Hyper(A)	0.0370 ± 0.00	−0.0000 ± 0.00	0.906 ± 0.00	0.907 ± 0.00	0.893 ± 0.00	**0.937 ± 0.00**
Hyper(G)	−0.019 ± 0.00	−0.0000 ± 0.00	0.884 ± 0.00	0.875 ± 0.00	0.850 ± 0.00	**0.911 ± 0.00**
Hyper(I)	**0.770 ± 0.00**	−0.000 ± 0.00	0.730 ± 0.00	0.745 ± 0.00	0.768 ± 0.00	0.720 ± 0.00
RBF(A)	−0.439 ± 0.00	−0.000 ± 0.00	**0.710 ± 0.00**	0.199 ± 0.00	0.482 ± 0.00	0.408 ± 0.00
RBF(G)	−0.449 ± 0.00	−0.000 ± 0.00	**0.702 ± 0.00**	0.164 ± 0.00	0.474 ± 0.00	0.402 ± 0.00
Avg. rank	5.45	4.91	2.32	3.86	2.68	**1.77**
Avg. rank real	5.45	4.64	2.73	4.18	2.36	**1.64**
Avg. rank synth.	5.45	5.18	**1.91**	3.55	3.00	**1.91**

Despite being regarded as the state-of-art, ARF struggled to outperform HT, a substantially simpler algorithm. This finding validates HT as a solid baseline since, even in some

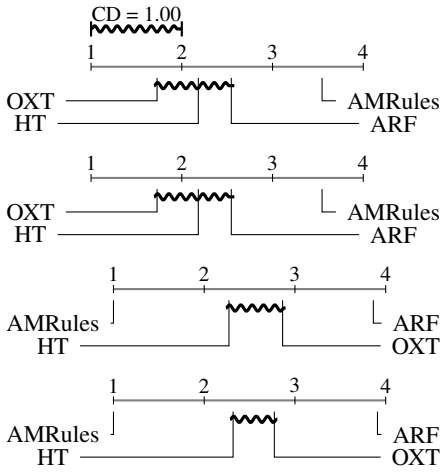


Figure 18 – Friedman test and Nemenyi post-hoc test results. From top to bottom: RMSE, R2, Memory, and Time. We removed the simplest baselines from the comparison (Dummy and PAR), as they are expected to always be the most inaccurate, lightweight, and fastest contenders, due to their simplicity.

non-stationary datasets, it could be competitive or even better. Furthermore, according to both Friedman-Nemenyi tests, HT was ranked in the second position and ARF in the third. However, no statistically significant differences were observed between the predictive performance of OXT, HT, and ARF.

In fact, OXT and HT obtained the same ranking position in predictive performance when considering the synthetic datasets used in our experiments (Tables 18 and 19). Specifically, the RBF regression datasets, first introduced by Gomes et al. (GOMES *et al.*, 2020), present a large gap in RMSE between HT and the remaining algorithms, even though HT has no explicit mechanisms to deal with concept drifts. We argue that, in such cases, HT kept creating splits to reflect the changes in the data. This fact, aligned with the linear models at the leaves, made HT the most accurate algorithm in the RBF datasets.

However, as the analysis of the memory footprint and running time show (Tables 20 and 21, respectively), a single tree became more computationally costly than the ensembles in the RBF datasets. If we only consider HT and OXT, there are even more cases where the running time or memory usage of a single tree surpasses the ensemble. AMRules, ARF, and OXT, on the other hand, reacted to the concept drifts and reset their inner states, as the reduced memory footprint indicates. Figure 19 illustrates the mentioned behaviors on the RBG(A) dataset.

Despite the apparent advantage of HT when merely considering the RMSE and R^2 rankings, we can see that a single tree was worse than ARF and OXT on real-world data regarding predictive performance. The same is true for the majority of synthetic datasets. OXT was the best-ranked algorithm in all scenarios, including real and synthetic data and stationary and non-stationary datasets.

The remaining regressors performed poorly in the majority of the cases, especially in

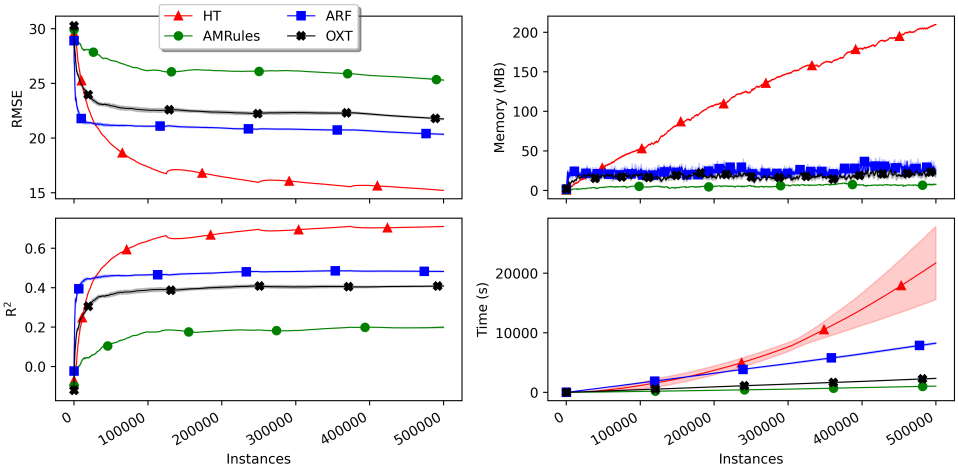


Figure 19 – Performance of AMRules, HT, ARF, and OXT on the RBF(A) dataset. As HT has no concept drift adaptation capabilities, the tree structure keeps growing and becomes more computationally costly than the ensembles.

Ailerons, Elevators, and Puma32H. In these datasets, all methods, except for PAR, performed very similarly regarding RMSE and R^2. We believe this is related to the distribution of the targets since, as shown in Table 17, their mean and standard deviation revolve around 0. Comparable results were reported in Gomes et al. (GOMES *et al.*, 2020) for some of the mentioned datasets. Similarly, the datasets with the highest mean and standard deviation values also yielded the highest obtained errors: CalHousing, Metro8L, and Metro16H.

Table 20 – Memory (MB) results

Dataset	PAR	Dummy	HT	AMRules	ARF	OXT
Abalone	0.025 ± 0.00	**0.023 ± 0.00**	0.733 ± 0.08	0.320 ± 0.00	11.546 ± 0.97	2.358 ± 0.32
Ailerons	0.0310 ± 0.00	**0.0280 ± 0.00**	6.116 ± 1.02	3.276 ± 0.60	129.944 ± 1.78	73.304 ± 2.14
Bike	0.0250 ± 0.00	**0.023 ± 0.00**	1.662 ± 0.17	0.367 ± 0.04	27.202 ± 4.36	11.352 ± 2.47
CalHousing	0.023 ± 0.00	**0.022 ± 0.00**	3.444 ± 0.30	0.544 ± 0.17	9.460 ± 1.12	3.258 ± 0.27
Elevators	0.025 ± 0.00	**0.024 ± 0.00**	5.074 ± 0.86	1.436 ± 0.25	145.286 ± 0.65	36.130 ± 1.54
House8L	0.023 ± 0.00	**0.022 ± 0.00**	3.850 ± 0.33	0.741 ± 0.07	149.844 ± 8.87	25.188 ± 3.05
House16H	0.025 ± 0.00	**0.023 ± 0.00**	8.290 ± 0.37	1.502 ± 0.31	155.658 ± 30.57	32.554 ± 2.53
Metro	0.036 ± 0.00	**0.027 ± 0.00**	1.932 ± 0.11	0.243 ± 0.04	105.994 ± 23.56	6.548 ± 1.27
Pol	0.036 ± 0.00	**0.032 ± 0.00**	6.004 ± 0.25	1.227 ± 0.38	33.324 ± 1.14	38.332 ± 5.63
Wind	0.035 ± 0.00	**0.028 ± 0.00**	4.978 ± 0.08	0.664 ± 0.04	112.756 ± 9.35	14.842 ± 4.65
Wine	0.025 ± 0.00	**0.023 ± 0.00**	1.756 ± 0.18	0.703 ± 0.07	41.556 ± 5.03	6.580 ± 0.67
Friedman	0.023 ± 0.00	**0.022 ± 0.00**	29.684 ± 0.68	2.960 ± 0.12	115.908 ± 33.35	80.514 ± 10.34
Puma8NH	0.023 ± 0.00	**0.021 ± 0.00**	2.800 ± 0.23	0.721 ± 0.10	83.556 ± 14.91	11.008 ± 1.62
Puma32H	0.030 ± 0.00	**0.027 ± 0.00**	10.174 ± 0.97	4.420 ± 0.63	141.002 ± 0.48	43.312 ± 4.31
Friedman(LEA)	0.005 ± 0.00	**0.004 ± 0.00**	29.910 ± 0.00	2.070 ± 0.00	127.656 ± 30.98	38.630 ± 2.73
Friedman(GRA)	0.005 ± 0.00	**0.004 ± 0.00**	27.190 ± 0.00	1.950 ± 0.00	108.710 ± 14.94	49.418 ± 6.26
Friedman(GSG)	0.005 ± 0.00	**0.004 ± 0.00**	27.690 ± 0.00	2.120 ± 0.00	101.126 ± 11.34	42.240 ± 4.92
Hyper(A)	0.005 ± 0.00	**0.004 ± 0.00**	81.050 ± 0.00	0.409 ± 0.00	324.623 ± 445.89	102.094 ± 11.11
Hyper(G)	0.005 ± 0.00	**0.004 ± 0.00**	83.570 ± 0.00	1.280 ± 0.00	722.978 ± 96.99	123.940 ± 9.93
Hyper(I)	0.005 ± 0.00	**0.0039 ± 0.00**	66.710 ± 0.00	0.619 ± 0.09	94.928 ± 22.55	2.042 ± 0.11
RBF(A)	0.008 ± 0.00	**0.006 ± 0.00**	209.620 ± 0.00	7.574 ± 0.62	25.360 ± 5.07	21.652 ± 4.36
RBF(G)	0.008 ± 0.00	**0.006 ± 0.00**	218.210 ± 0.00	7.530 ± 0.00	24.750 ± 3.46	21.362 ± 1.54
Avg. rank	2.00	**1.00**	4.27	3.00	5.86	4.86
Avg. rank real	2.00	**1.00**	4.09	3.00	5.91	5.00
Avg. rank synth.	2.00	**1.00**	4.45	3.00	5.82	4.73

A considerable disparity is observed when analyzing the memory footprint of our method. As shown in Table 20, in all datasets evaluated, except Pol, OXT employed a substantially inferior amount of memory in comparison to ARF. This difference is further pronounced in the non-stationary datasets, e.g., in the Hyper(I) dataset, OXT requires, on average, approximately 2MBs,

whereas ARF utilizes around 95MBs. Naturally, the baseline algorithms were mainly associated with smaller footprints due to their simplicity. According to the Friedman-Nemenyi test (Figure 18), our method is not statistically different from HT, attesting to its efficiency. As opposed to that, ARF was ranked in the lowest ranking.

Similar behavior is noticed for the running time (Table 21) where OXT outpaces its direct competitor algorithm in all datasets evaluated. As prominent results, we can highlight the running times obtained from all three variants of the Hyper dataset where OXT often managed to be more than ten times faster than ARF. Surprisingly, there are cases where OXT is more efficient than a single decision tree (HT), as seen in the datasets with concept drift. As HT has no mechanism for dealing with concept drifts, the tree model continues to be expanded even after drifts occur. On the other hand, OXT resets its base models when drifts are detected, thus reducing the ensemble size and enabling faster executions. In the stationary counterparts, however, OXT is 2 or 3 times slower than HT in most cases. When analyzing the Friedman-Nemenyi, Figure 18, interchangeable conclusions can be drawn, as our method is significantly more efficient than ARF but not different from HT.

Table 21 – Time (s) results

Dataset	PAR	Dummy	HT	AMRules	ARF	OXT
Abalone	1.313 ± 0.04	**0.976 ± 0.08**	2.082 ± 0.05	1.949 ± 0.01	45.433 ± 0.85	9.272 ± 0.11
Ailerons	6.809 ± 0.16	**5.161 ± 0.01**	28.873 ± 2.21	21.550 ± 0.37	386.010 ± 16.48	98.374 ± 1.92
Bike	4.227 ± 0.02	**2.967 ± 0.00**	11.149 ± 0.71	8.810 ± 0.28	201.416 ± 10.14	60.484 ± 2.26
CalHousing	4.233 ± 0.02	**2.866 ± 0.02**	16.585 ± 0.45	8.869 ± 0.50	207.498 ± 3.60	42.943 ± 0.46
Elevators	5.027 ± 0.11	**3.595 ± 0.00**	20.874 ± 1.58	13.807 ± 0.14	455.661 ± 3.16	69.850 ± 1.56
House8L	4.764 ± 0.15	**3.167 ± 0.00**	19.303 ± 0.88	10.632 ± 0.14	624.719 ± 44.31	74.512 ± 5.76
House16H	6.407 ± 0.01	**4.607 ± 0.02**	37.344 ± 0.48	19.523 ± 0.43	860.593 ± 51.25	107.874 ± 2.95
Metro	20.860 ± 0.07	**12.835 ± 0.02**	24.997 ± 0.80	14.417 ± 0.31	1036.938 ± 93.47	175.541 ± 13.01
Pol	8.524 ± 0.02	**6.681 ± 0.01**	26.203 ± 0.85	19.862 ± 0.30	219.806 ± 5.90	123.894 ± 6.03
Wind	2.987 ± 0.00	**1.985 ± 0.00**	8.941 ± 0.02	4.314 ± 0.02	164.089 ± 10.05	22.974 ± 1.67
Wine	1.234 ± 0.00	**0.854 ± 0.00**	3.443 ± 0.21	2.840 ± 0.01	76.047 ± 1.38	13.132 ± 0.71
Friedman	21.244 ± 0.08	**14.652 ± 0.04**	483.768 ± 18.00	83.138 ± 2.93	4201.537 ± 217.64	814.193 ± 40.32
Puma8NH	1.679 ± 0.01	**1.132 ± 0.00**	5.750 ± 0.23	3.726 ± 0.12	178.039 ± 23.75	20.855 ± 0.83
Puma32H	3.470 ± 0.03	**2.622 ± 0.05**	21.395 ± 1.39	12.276 ± 0.97	270.810 ± 5.13	51.988 ± 1.70
Friedman(LEA)	12.069 ± 0.01	**6.213 ± 0.01**	440.665 ± 27.27	71.590 ± 0.81	5736.182 ± 239.46	840.524 ± 62.10
Friedman(GRA)	12.794 ± 0.04	**6.305 ± 0.02**	453.617 ± 1.84	59.129 ± 0.06	2670.361 ± 87.29	526.304 ± 21.03
Friedman(GSG)	12.144 ± 0.02	**6.184 ± 0.01**	391.634 ± 0.48	61.082 ± 0.20	2850.983 ± 193.38	513.556 ± 23.89
Hyper(A)	62.327 ± 0.14	**31.762 ± 0.11**	6814.025 ± 67.67	223.004 ± 0.85	46713.597 ± 4566.88	4899.106 ± 447.36
Hyper(G)	63.484 ± 0.08	**32.521 ± 0.09**	7024.612 ± 114.74	229.853 ± 0.45	31871.921 ± 1563.84	3715.063 ± 176.41
Hyper(I)	62.996 ± 0.19	**32.172 ± 0.05**	4999.194 ± 114.30	239.656 ± 3.69	21599.186 ± 2295.65	1245.812 ± 36.76
RBF(A)	83.308 ± 0.36	**48.259 ± 0.06**	21712.278 ± 6143.04	1044.805 ± 5.66	8248.648 ± 119.48	2345.118 ± 28.42
RBF(G)	148.937 ± 25.79	**69.309 ± 30.19**	17345.020 ± 252.87	826.134 ± 1.68	7557.625 ± 143.39	2514.738 ± 37.67
Avg. rank	2.05	**1.00**	4.32	2.95	5.91	4.77
Avg. rank real	2.09	**1.00**	4.00	2.91	6.00	5.00
Avg. rank synth.	2.00	**1.00**	4.64	3.00	5.82	4.55

It is worth noting that as reported by Mastelini and Carvalho (MASTELINI; CARVALHO, 2021), TE-BST is more efficient than E-BST regarding running time and used memory. OXT is able to outperform ARF even when the latter ensemble is using an efficient AO. Even more pronounced disparities between the memory footprint and running time should be observed if ARF applies the traditional E-BST as its template AO.

6.6.2 Saturation study

In this section, we perform a saturation study on two datasets with distinctive characteristics: Friedman(GRA) and CalHousing. More precisely, we evaluate the effect of adding more base learners to the ensemble. Starting from 10 trees, we have sequentially added 10 trees to each model until the maximum of 100 trees was reached.

Additionally, we also assessed the change in performance when linear models are replaced by the mean prediction at the tree leaves. The Friedman(GRA) and CalHousing datasets were selected since Friedman(GRA) is non-stationary and OXT was slightly superior to ARF. On the other hand, in the stationary CalHousing dataset, ARF had the predictive upper hand over OXT. For the saturation study, we rely on heatmaps, illustrated by Figure 20. In these charts, the x axis represents the number of processed instances, and each row in y represents the number of trees used by ARF or OXT. The color grading is the mean percentual variation of RMSE among all the experiment repetitions, in relation to the default number of trees used in our experiments (20). For that reason, the row representing the ensembles with 20 trees (whose label is in **green**) always shows a 0% change in RMSE.

As depicted in Figure 20, OXTs saturate faster than ARF. Namely, the addition of more base learners does not highly impact the performance of the OXTs, as a small number of trees managed to perform very similarly to larger forests. We perceive this as an indication that OXTs ended up building more homogeneous models, resulting in rapid convergence. As for the ARFs, a tenuous but rather visible difference is perceivable when the number of trees increases.

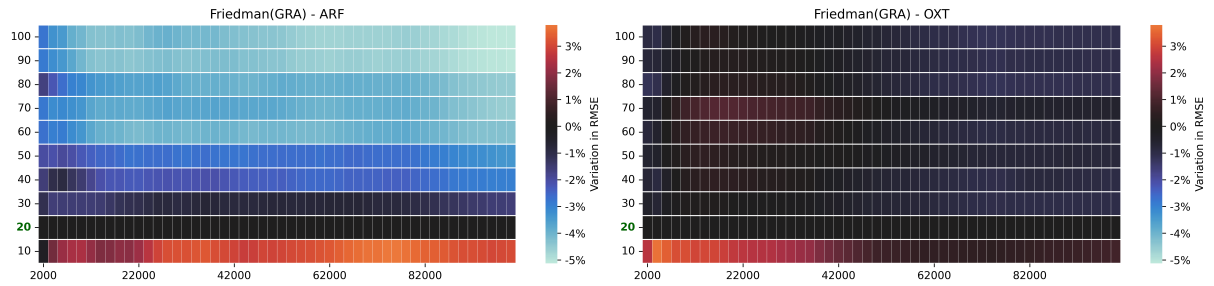


Figure 20 – Assessing the error saturation by adding trees to the algorithms ARF and OXT on the Friedman(GRA) dataset. The number of trees employed, the value obtained for the evaluation measure, and the number of instances processed are presented on the left y-axis, right y-axis, and x-axis, respectively.

By carefully inspecting the presented performances, we notice that ARF requires roughly 60 trees to match the performance obtained by OXT with only ten trees. In order to further examine their differences, we also compare their memory footprint and running time in Figure 21. Despite yielding overlapping performances, the computational complexity contrast is perceptible where OXT is remarkably more efficient and lightweight.

In terms of saturation, very similar conclusions may be drawn from the CalHousing dataset (Figure 22). However, a different behavior was observed in the predictive performance,

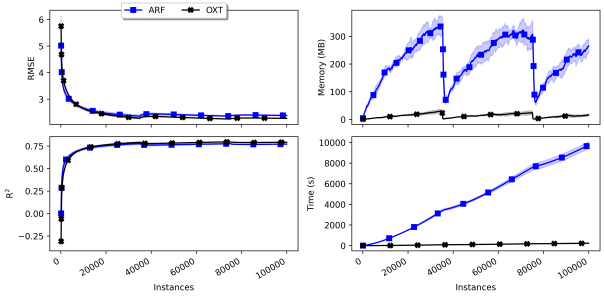


Figure 21 – Comparing the predictive performance and computation resource usage on the Friedman(GRA) dataset using ARF (60 trees) and OXT (10 trees).

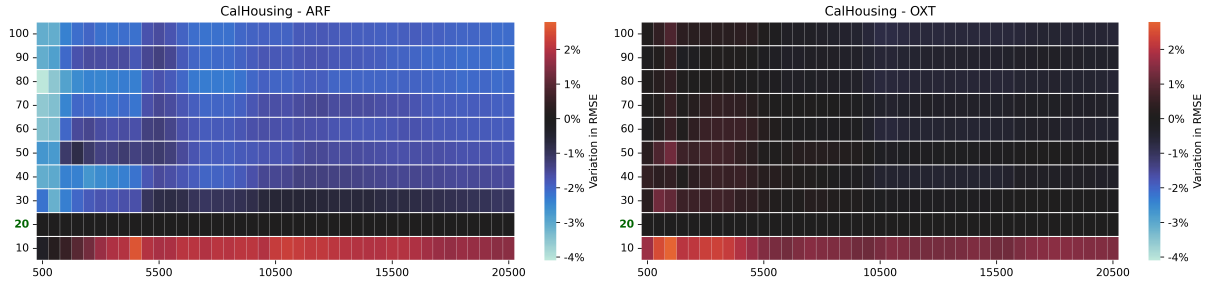


Figure 22 – Assessing the error saturation by adding trees to the algorithms ARF and OXT on the Calhousing(GRA) dataset. The number of trees employed, the value obtained for the evaluation measure and the number of instances processed are presented on the left y-axis, right y-axis, and x-axis, respectively.

as ARF had the upper hand regardless of the number of trees. For brevity, we do not include a detailed inspection of the performance differences. Nonetheless, we argue that, since CalHousing contains a relatively small number of features and instances, the size of the dataset may present a challenge in the early stages of learning. We believe that is due to the random nature of OXTs, which may result in a cold start problem that primarily affects the models in the leaves.

When conventional regression trees are used as base learners (mean prediction in the leaves), the difference in performance saturation is more evident. As shown in Figure 23, the addition of models to the ARF is evidently more beneficial, whereas a less noteworthy change is seen in the OXT algorithm.

Furthermore, when comparing the performance differences between the ensembles presented in Figures 20 and 23, we noticed that employing adaptive linear models is associated to superior results. Likewise, the RMSE decreases considerably faster with the linear models. We hypothesize that conventional regression trees may be more susceptible to mistakes in the first iterations since the mean of a relatively large subset is being used as predictions. As opposed to that, linear models are capable of amending such mistakes by performing an actual prediction.

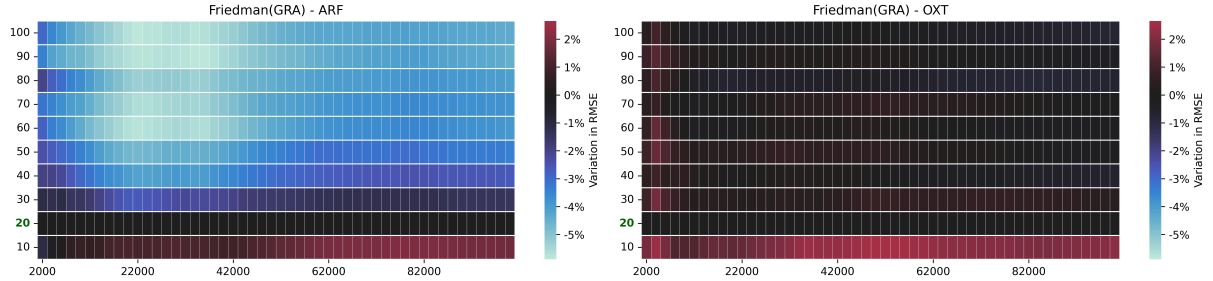


Figure 23 – Ablation study: assessing the error saturation on the Friedman GRA dataset when using regression trees as base learners. The predictions of the were aggregated using the mean predicted value. The number of trees employed, the value obtained for the evaluation measure and the number of instances processed are presented on the left y-axis, right y-axis, and x-axis, respectively.

6.6.3 Understanding the difference in performance between ARF and OXT: the characteristics of the forest

To further comprehend our experiments, we investigate the properties of the forests generated using two datasets: Elevators and Friedman(GRA). More precisely, we compare the number of nodes, number of leaves, average tree height, and total re-sampling weight using 20 trees in both models. The obtained results are presented in Figure 24.

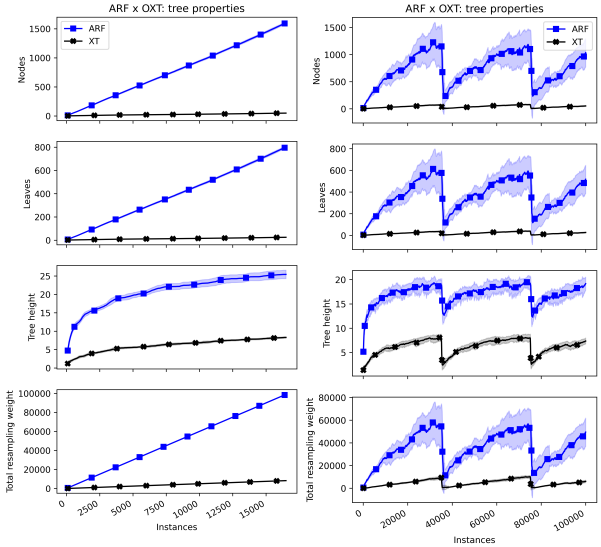


Figure 24 – Comparing the structures and weight amount observed by the forests generated using ARF and OXT, the Elevators (left) and Friedman(GRA) (right) datasets, and 20 trees. From top to bottom: number of nodes, leaves, tree height, and total (re-)sampling weight.

In the Elevators dataset, it is noticeable that ARF presents a growing tendency, where the size of all properties increases as more data is processed. Contrary to that, OXT splits considerably less often, resulting in shallower and memory-wise smaller models.

These properties are seemly beneficial since the predictive performance remains unaffected, whereas the memory footprint and the running time are severely reduced. As a downside,

as observed in the last section, the constituent base learners become less diverse.

Intuitively, obtaining shallow trees is a direct result of the selected splitting strategy and the number of processed instances. Since ARF employs a deterministic strategy to select the best split candidates, we expect its splits to present marginally superior quality than the random strategy employed by OXT. Hence, the split candidates in ARF have an increased chance of being selected and performed. Moreover, as aforementioned, ARF extends the original online bagging algorithm (OZA; RUSSELL, 2001b) by increasing the instance re-sampling rate, i.e., effectively increasing the λ parameter as in Leveraging Bagging (BIFET; HOLMES; PFAHRINGER, 2010). OXT uses sub-sampling, which makes its trees process much fewer instances, as indicated in Figure 24.

As a possible countermeasure, besides relaxing the split decision restrictions, as we did (refer to subsection 6.5.4 for details on hyperparameters used in OXT), the sub-bagging rate could be increased (we kept this value fixed at 50%, as previously mentioned). Hence, each tree in the OXT forest would process more instances, perform more split decisions, and, potentially, become more diverse. However, it is worth mentioning that this change can result in higher computational complexity.

Similarly, an identical behavior is observed in the Friedman(GRA) dataset. However, sudden decreases are present in this case due to the concept drifts countermeasure mechanism, which might completely remove trees from the forest.

As observed in the previous subsection, employing adaptive linear models instead of the mean prediction often provides superior results. Thus, as a positive side-effect of its splitting criterion, combined with sub-bagging, OXT balances the computational cost and performance associated with its linear models, thus building relatively small but powerful regression trees.

6.7 Final remarks

In this work, we have proposed a novel online ensemble of regression trees, namely OXT, which adapts the well-established extra trees algorithm to online learning. More specifically, we proposed a novel attribute observer, which defines random split points by keeping an initial and small buffer of observations. This buffer, once used for determining the input range, is discarded by the AO. OXT also relies on sub-bagging to further decrease the computational costs and boost predictive performance.

When compared to the current state-of-the-art ARF, our proposed algorithm is significantly more efficient. We believe this result is a combination of the splitting criterion employed by OXT, which builds reasonably smaller trees, and the instance sampling strategy.

Despite relying on a random split strategy (ARF, AMRules, and HT use deterministic strategies for making splits), OXT is capable of delivering superior or competitive predictive

performance in most of the evaluated cases. Thus, our proposed model is capable of balancing the trade-off between computational complexity and performance by imposing a strict yet lightweight splitting criterion alongside adaptive linear models in the leaves.

As possible future work, we would like to explore different hyperparameter configurations which would allow the trees to split more often and, possibly, yield better results. Likewise, we could further investigate our proposed algorithm by including real-world datasets with concept drift. Finally, we would also like to address the performance of OXT in cases where label delay is present, that is, applications where the labels of new data are only available after a non-negligible latency (GOMES *et al.*, 2022).

CHAPTER

7

SWINN: EFFICIENT NEAREST NEIGHBOR SEARCH IN SLIDING WINDOWS USING GRAPHS

Publication information: the contents of this chapter were sent for evaluation at Elsevier's Information Fusion, and they are publicly available on the SSRN platform.

Authors: Saulo Martiello Mastelini, Bruno Veloso, Max Halford, Andre C. P. L. F. de Carvalho, and João Gama.

7.1 Abstract

Nearest neighbor search (NNS) is one of the main concerns in data stream applications, since similarity queries can be used in multiple scenarios. Online NNS is usually performed on a sliding window by lazily scanning every element currently stored in the window. This paper proposes Sliding Window-based Incremental Nearest Neighbors (SWINN), a graph-based online search index algorithm for speeding up NNS in potentially never-ending and dynamic data stream tasks. Our proposal broadens the application of online NNS-based solutions, as even moderately large data buffers become impractical to handle when a naive NNS strategy is selected. SWINN enables efficient handling of large data buffers by using an incremental strategy to build and update a search graph supporting any distance metric. Vertices can be added and removed from the search graph. To keep the graph reliable for search queries, lightweight graph maintenance routines are run. According to experimental results, SWINN is significantly faster than performing a naive complete scan of the data buffer, while keeping competitive search recall values. We also apply SWINN to online classification and regression tasks, and show that our proposal is effective against popular online machine learning algorithms.

7.2 Introduction

Nearest Neighbor Search (NNS) is an essential component of many computer science applications (AUMÜLLER; BERNHARDSSON; FAITHFULL, 2020). The notion of proximity can be extended to multiple interpretations and dealt with differently by distinct research communities. Proximity is also correlated to the concept of learning, e.g., humans learn by replicating experiences and extrapolating from known similar patterns. In machine learning (ML), proximity is the pivot of multiple learning paradigms and algorithms: k-Nearest Neighbors (k-NN), Support Vector Machines, and k-Means, to name a few. Recent works have confirmed the importance of NNS to ML, such as (DOMINGOS, 2020), which suggested a link between Deep Neural Networks and NNS. Thus, more evidence highlighting the importance of NNS is available nowadays.

Nonetheless, using a naive approach for dealing with NNS can rapidly become impractical, even with moderately sized datasets, e.g., a brute-force strategy or a complete Linear Scan (LS) (LS) of the data. Therefore, in many applications, more efficient NNS approaches are needed. ML research leverages discoveries from other research areas to enhance NNS, thus enabling large-scale computing (DATAR *et al.*, 2004; JEGOU; DOUZE; SCHMID, 2010; ANDONI *et al.*, 2015; MATSUI *et al.*, 2018; SHIMOMURA *et al.*, 2021). A wide variety of solutions have been proposed for batch applications, i.e., the training data is available from the start and does not change over time. Still, NNS is an open challenge, and novel solutions keep being proposed over the years. The most usual strategy is to devise auxiliary data structures, referred to as search indices, for which performing NNS is faster than relying on an LS of the data. Such search indices may provide either exact or approximate search results. A reference tool for NNS benchmarking was introduced in (AUMÜLLER; BERNHARDSSON; FAITHFULL, 2020), where multiple NNS techniques are compared.

NNS is also relevant for online ML applications. In these scenarios, data arrives continuously and might change its generative distribution over time. Typically, online k-NN algorithms keep a buffer of the most recent instances where they perform the NNS. This data buffer is updated using a First in, First out (FIFO) strategy, i.e., a sliding window. To the best of our knowledge, search via LS is the prevailing strategy in streaming applications, with the user controlling the data buffer length. Hence, considering the buffer length L, the cost to perform a single query is $O(L)$. On the other hand, search queries are exact. Reducing the cost of an LS could be helpful even if it reaches approximate results.

In this study, we want to stress the relevance of k-NN-based algorithms in streaming applications (BARROS; SANTOS; BARDDAL, 2022). Although k-NN is a straightforward lazy algorithm, it poses a strong baseline or even the best ML algorithm for multiple tasks. Moreover, k-NN is also a robust learning algorithm, as it supports user-defined distance measures and has few hyperparameters for adjustment. This last aspect is especially relevant in online ML due to the dynamic nature of the learning tasks and the inherent difficulty of adjusting

hyperparameters online. Besides, the vanilla k-NN algorithm is naturally robust against concept drift in the online setting. By keeping a buffer of the most recent instances, outdated concepts are automatically forgotten, whereas new concepts are put to the forefront. Some online k-NN variants implement long-term memory schemes and sophisticated mechanisms for dealing with concept drift (LOSING; HAMMER; WERSING, 2018; LOSING *et al.*, 2020). Still, these solutions do not employ efficient strategies for speeding up NNS.

Some effort has been made to create incremental search indices and, thus, speed up NNS time. Examples include kd-Trees (JO; SEO; FEKETE, 2018; CAI; XU; ZHANG, 2021), Product Quantization and related techniques (XU; TSANG; ZHANG, 2018; LIU *et al.*, 2021). Nonetheless, such solutions are limited to specific distance measures and might offer limited support for removing data from the search index.

Graph-based NNS has risen as the state-of-the-art in recent years (AUMÜLLER; BERNHARDSSON; FAITHFULL, 2020; SHIMOMURA *et al.*, 2021). Besides working with generic distance or similarity metrics, graphs can be naturally expanded with new nodes, an appealing feature to online ML applications. Nonetheless, to the best of our knowledge, frequent element removal and its impact on the search graph structure are still to be explored. This paper proposes Sliding Window-based Incremental Nearest Neighbors (SWINN) to handle approximate NNS in sliding windows. Our proposal is inspired by NN-Descent (DONG; MOSES; LI, 2011), one of the most popular graph-based approximate NNS strategies for batch data. SWINN is significantly faster than an LS of the sliding window when L is sufficiently large while keeping competitive search recall.

Our main contributions can be summarized as follows:

- We propose Sliding Window-based Incremental Nearest Neighbors (SWINN) for handling NNS in sliding windows;
- We compare SWINN against the current state-of-the-art online k-NN models using a comprehensive and extensive synthetic setup. We show that our proposal can deliver comparative search recall results while being significantly faster than an LS of the data buffer;
- We study how each hyperparameter of SWINN impacts its running time, memory footprint, and search recall. We also suggest a set of default hyperparameter values to use in general tasks and provide guidelines to select good hyperparameter value combinations if tuning is required;
- We study in which situations SWINN might be preferable to a LS and in which situations the opposite holds;
- We perform case studies comparing SWINN against popular online ML models in classification and regression tasks, showing more evidence to support our proposal's effectiveness.

The remaining of this manuscript is organized as follows. Section 7.3 presents previous works related to this research and the theoretical foundations needed to support our proposal and facilitate its understanding. We introduce SWINN and its main aspects in section 7.4. We present our synthetic setup to compare SWINN against the current prevalent solution, i.e., an LS of the sliding window, in section 7.5. Next, we discuss the obtained results using the controlled data in section 7.6 and provide guidelines for selecting hyperparameter values. Complementary results are available in Appendix A. We perform case studies comparing SWINN and LS in classification and regression tasks against popular online ML algorithms in section 7.7. Finally, we present our final considerations and future work directions in section 7.8.

7.3 Background

This section presents the related work closest to our proposal and the theoretical foundations needed to understand better how SWINN works.

7.3.1 Related work

Multiple search index algorithms were proposed in the search for more efficient NNS strategies, primarily for batch applications. Some of these search indices were also adapted to allow incremental point addition and, in rare cases, element removal. The usage of partition trees is a frequent trend. In particular, the kd-Tree (FRIEDMAN; BENTLEY; FINKEL, 1977) and Ball-tree (OMOHUNDRO, 1989) models are widely employed to create query structures for NNS. Ball-trees can deal with high-dimensional data, whereas kd-trees are better suited for datasets with a small or moderate number of dimensions. A few attempts to adapt kd-Tree to incremental environments were proposed in the last years (JO; SEO; FEKETE, 2018; CAI; XU; ZHANG, 2021). These adaptations can deal with data insertion and element removal. Nonetheless, frequent insertion/removal of elements (as in a sliding window regimen) makes these tree-based solutions impractical to our application setup due to the frequent need to re-balance the tree structures. Besides, kd-Trees are not well-suited to high-dimensional scenarios. Ball-trees could be used instead. However, their construction can become costly, even in batch-based applications. Another limitation of existing partition tree solutions is that they cannot work with arbitrary distance metrics.

A popular alternative trend in batch-based NNS is Locality Sensitive Hashing (LSH) (DATAR *et al.*, 2004; ANDONI *et al.*, 2015). Unlike traditional hashing techniques, LSH aims to map similar inputs to the same hash table position. Multiple families of mapping functions suited to different distance metrics were proposed throughout the years. Effectively, LSH acts as a discretization technique: query points will be mapped to the same positions as their nearest neighbors with high probability. LSH has been effectively used in multiple real-world applications (ANDONI *et al.*, 2015). Although LSH's projection scheme is naturally incremental, LSH

techniques cannot be easily set up online due to the high dependence on the correct choice of hyperparameter values. Non-stationary scenarios might make this problem even more evident. Besides, data projection might be too costly to perform constantly. Lastly, LSH solutions also cannot work with arbitrary distance metrics.

Random Partition Trees (RPT) were effectively applied to real-world NNS problems by combining aspects from the two previous NNS solutions. One of the most popular versions of RPTs is implemented in the Spotify-backed library, Annoy (AUMÜLLER; BERNHARDSSON; FAITHFULL, 2020). Annoy creates ensembles of oblique trees whose partitions are hyperplanes equidistant to two randomly sampled points. Therefore, the RPTs use a partition scheme similar to many LSH solutions rather than applying axis-aligned splits like kd-Trees. RPTs can be independently created because the partitions do not depend on data-driven statistics. Moreover, tree construction is comparatively cheaper than in the vanilla kd-Tree algorithm. On the other hand, RPTs are also limited to a restricted group of distance metrics and are static by design. Once the RPT forest is created, it should be used to perform all search queries. Element addition and removal are not trivial, as we deal effectively with multiple search indices, i.e., the individual trees, rather than a single search index.

Vector Quantization (VQ) is another strategy used for NNS (JEGOU; DOUZE; SCHMID, 2010; MATSUI *et al.*, 2018). This family of solutions can efficiently deal with high-dimensional data but can be comparatively less accurate than the other presented strategies. The efficiency and performance trade-off is dependent on the correct hyperparameter choice. VQ-based solutions are also limited to specific distance metrics. The most popular VQ strategy is Product Quantization (PQ), which relies on batch clustering to create data partitions and encode the original data using the created partitions' information. Variations of VQ, including Product Quantization, have been proposed for dealing with online learning scenarios (XU; TSANG; ZHANG, 2018; LIU *et al.*, 2021). The online PQ strategy can be applied to a sliding window learning regimen but might be too restrictive regarding distance measures.

Graph-based methods have become a synonym for NNS effectiveness in the last few years (SHIMOMURA *et al.*, 2021). Among the existing solutions, small-world graph-based methods (MALKOV; YASHUNIN, 2018; SHIMOMURA *et al.*, 2021) have received special attention from the research community. Nonetheless, the cost of graph construction renders this kind of solution impractical in online learning scenarios. Still, early baselines for graph-based NNS have interesting properties that enable their adaptation to online NNS. In particular, the pivotal NN-Descent algorithm (DONG; MOSES; LI, 2011) is especially appealing because its construction is inherently incremental. NN-Descent also works with arbitrary distance metrics and could be adapted to handle frequent element addition and removal. Our proposal, SWINN, is inspired by the NN-Descent search index-building algorithm. Nonetheless, we extend the original algorithm to enable performing local changes in the search index. Such changes are necessary to add and remove new elements and to keep the graph reliable for search queries.

7.3.2 Preliminaries

Next, we present definitions used throughout the paper that help to describe the functioning of SWINN. We start by formalizing the concept of a graph.

Definition 1: a weighted graph, G, is defined by the tuple $G = (V, E)$, where V represents the set of vertices and $E = \{(o, d, w) \mid o \in V, d \in V, w \in \mathbb{R}, \text{and } o \neq d\}$ is the edge set. In the cases where the weight information is irrelevant, we will use (o, d) as a shorthand for (o, d, w).

If the edges in E are directed, then $(o, d) \neq (d, o)$. If the ordering of the edges is irrelevant, the graph is undirected. SWINN uses directed graphs weighted by the distance between the instances in the sliding window. For such, we need to define the concept of the direct neighborhood.

Definition 2: the direct neighborhood, $N_v \subset V$, of a vertex $v \in V$ is defined as $N_v = \{n \in V \setminus \{v\} \mid (v, n) \in E\}$, i.e., the nodes for which v have direct edges.

In our setting, we consider a first-in, first-out (FIFO) data buffer with limited size, i.e., a sliding window. We denote by L the sliding window length, which is a user-defined parameter.

7.4 Sliding Window-based Incremental Nearest Neighbors

We propose **S**liding **W**indow-based **I**ncremental **N**earest **N**eighbor (SWINN), which is a modified version of the NN-Descent (DONG; MOSES; LI, 2011) algorithm suited for incremental learning scenarios. SWINN is based upon the assumption that "the neighbor of my neighbors might as well be my neighbor". Each instance in the sliding window data buffer is mapped to a vertex in the neighborhood graph. New vertices are added as new instances arrive, and the oldest instances (and their corresponding vertices) are dropped from the graph using the FIFO strategy. SWINN uses an iterative process to create and keep a NN graph. The high-level basic steps for graph construction are given as follows:

1. Start with a random neighborhood graph;
2. For each node in the search graph:
    a) Refine the current neighborhood by checking if there are better neighborhood options among the neighbors of the current neighbors;
3. If the total number of neighborhood changes is smaller than a given stopping criterion, then stop.

Although simple, the above-presented general lines for NNS graph construction are powerful in multiple aspects. First, there are no limitations in the type of distance measure used to build the search index, which is not the case when considering LSH, trees, or quantization-based search indices (JEGOU; DOUZE; SCHMID, 2010; MATSUI *et al.*, 2018). Second, the graph-building process is incremental by design. Moreover, unlike tree-based indexes, e.g., kd-trees (JO; SEO; FEKETE, 2018; CAI; XU; ZHANG, 2021), we can perform searches from any starting vertex, making index update procedures easier.

We can improve the above graph construction strategy by directly joining the neighbors of a given node. Instead of performing two graph hops, i.e., checking whether the neighbors' neighbors are better candidates than the current ones, we make all the neighbors of a given node consider each other as possible new neighbors. Hence, we perform a single traversal hop.

The proposed strategy to build the search graph was first introduced in the original NN-Descent paper (DONG; MOSES; LI, 2011). This strategy promotes neighborhood joins, i.e., vertices consider other vertices as potential new neighbors. We work with directed graphs; each vertex must have at most K direct neighbors. We use a capital letter to differentiate the number of neighbors used for graph building (K) and the number of neighbors used during search queries (k). The former value is fixed for SWINN's graphs, whereas the second may vary for each query. Although the number of direct neighbors, K, is fixed, we also need to consider the reverse neighborhood, as defined next.

Definition 3: the reverse neighborhood, $N'_v \subset V$, of a vertex $v \in V$ is defined as $N'_v = \{n_r \in V \setminus \{v\} \mid (n_r, v) \in E\}$, i.e., the vertices with direct edges to v.

Considering the reverse neighborhood, the number of edges might increase considerably to a value higher than K. As we work in a sliding window regimen, the maximum number of vertices the search graph will have is limited by the length of the sliding window (L), i.e., $L = |V|$. A vertex's maximum number of reverse neighbors is bounded by $L-1$. The input data define reverse neighborhood characteristics. Some works discussed different phenomena related to the placement of points in the search space and how these factors influence graph-based search indices (BRATIĆ *et al.*, 2018; BRATIĆ *et al.*, 2019). For instance, vertices with a high number of direct and reverse neighbors, i.e., the so-called hubs, might increase the chance of a search reaching local minima. Given the typically reduced number of instances considered in the sliding window learning regimen, discussing the mentioned influencing factors is out of the scope of the current work and will be addressed in future research.

Next, we define the total neighborhood as follows:

Definition 4: the total neighborhood, T_v, of a vertex $v \in V$ is defined as $T_v = N_v \cup N'_v$, i.e., the union between the direct and reverse neighbors of v.

When presenting the high-level description of the graph-building mechanism, we assumed that a set of instances was previously available. In online ML scenarios, learning occurs as soon as new instances arrive. SWINN has a `warm-up` period when instances are buffered, and no graph is built. During this stage, searches are performed using the LS for all the data. After the warm-up period, SWINN builds an initial graph and keeps updating the search index by adding and removing instances. From this point onward, only the NN graph is used to perform search queries.

We will detail in the following subsections each aspect of SWINN, including graph refinement, how to perform search queries using the NN graph, how to remove vertices from the graph, and possible ways to reduce the memory footprint of the search index. In the end, we will combine the description of each component to describe how the whole SWINN algorithm works.

7.4.1 Refinement

The refinement procedure sequentially exchanges edges to make the search index converge to the NN graph. For such, SWINN performs neighborhood joins considering combinations of reverse neighbors and direct neighbors related to a reference vertex, i.e., select all combinations among their reverse and direct neighbors for a given vertex.

Since the number of reverse neighbors a vertex can have is unbounded, the number of possible combinations may become too high. Limiting the number of neighbors joined at each refinement step can decrease costs. This strategy was proposed for NN-Descent, and we adopted it for SWINN. Specifically, SWINN takes a sample of the total neighborhood from a reference vertex when its total number surpasses a user-given threshold, $\texttt{max}_c$.

Even by limiting the number of vertices involved in neighborhood joins, NN-Descent can still perform redundant neighborhood change checks. This may happen when a previously attempted edge connection occurs during neighborhood joins. To reduce this problem and save processing time, NN-Descent keeps track of edges already tried by keeping binary flags for each vertex. Thus, each time a new direct neighbor is added to a vertex, a binary flag corresponding to the new neighbor is set to `true`. If this neighbor is used in a neighborhood join, its binary flag is set to `false`.

Neighborhood joins are performed by simultaneously exploring the following combinations: (1) both vertices involved in the join have their flags set to `true`; (2) the origin of the edge has a `true` flag, whereas the destination has a `false` flag.

These two constraints ensure that the left side of the join, i.e., the origin of the directed edge, was not previously involved in a neighborhood join. Note that reverse neighbors also have a

binary flag corresponding to the reference vertex. Although the binary flags add a slight memory overhead, they avoid trying to link vertices that were previously evaluated. Such as NN-Descent, SWINN applies an incremental version of this optimization.

Figure 25 presents an example illustrating how the refinement procedure works in SWINN. We start our example in step A, where an initial graph built with $K = 2$ is presented. In this state, the vertices are not necessarily linked to their NNs. In step B, the vertex green is the target vertex, and its neighbors (direct and reverse) are selected for the neighborhood join. In steps B, C, and D, the edges of vertices blue, gray, and red, respectively, are updated. In all the cases, better neighbors are selected by computing the distances from the vertex in focus (signaled by the red arrow) to the other vertices involved in the neighborhood join. SWINN deletes previous edges to make room for the new ones in case the new options are closer in the distance.

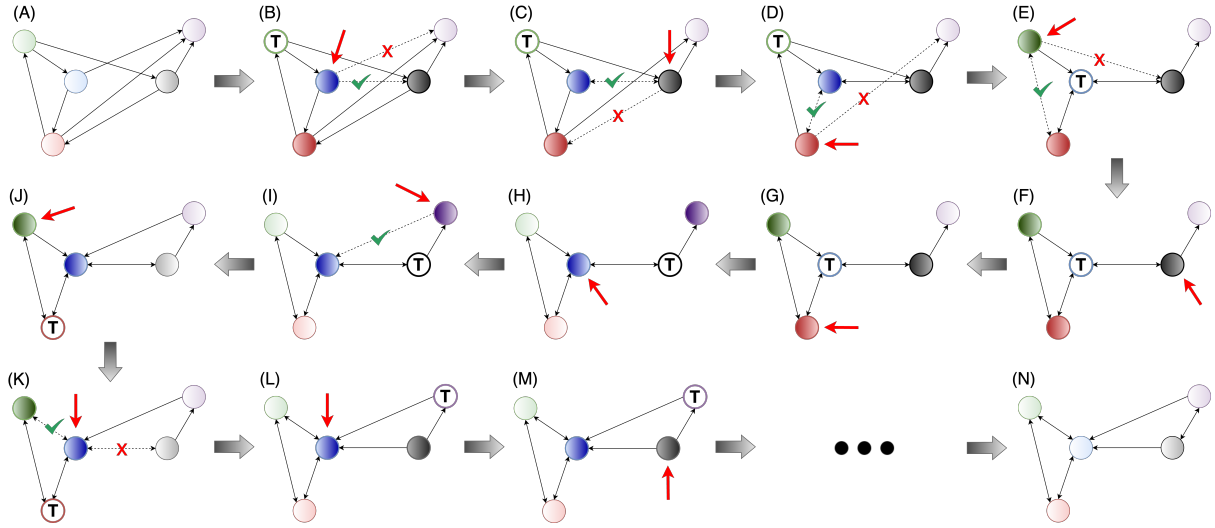


Figure 25 – An example of the SWINN refinement procedure. From steps A to M, one iteration of neighborhood refinement is illustrated. Step N shows the search index after the neighborhood refinement converges. Vertices with a bold **T** are the target during the refinement steps. Vertices with a darker color shade are the neighbors of the target vertex, i.e., the vertices involved in the neighborhood join. Arrows indicate the vertex whose neighbors are updated in each step.

The refinement proceeds by shifting the target vertex to blue in step E. Similarly, the neighborhood of vertex green is improved in step E. However, in steps F and G, the existing connections do not change, as gray and red are already connected to their best neighboring vertices. The same happens to blue when gray becomes the target vertex (step H). At this point, purple has only one reverse neighbor (gray) and no direct neighbors. Thus, in step I, purple adds a new edge to blue, as the latter is the only available option.

During step J, red becomes the new target. At this stage, the neighborhood does not change for green since blue is already its direct neighbor. Next, blue adds an edge to green in step K. SWINN does not limit reciprocal edges between vertices. The benefits of applying such

a constraint to the size of the edge set are surpassed by the additional actions needed to keep the NN connections as more data are monitored.

Notwithstanding, if it were not by the edge (purple, blue), in step K, two sub-graphs would be generated by removing the connection (blue, gray). This situation is problematic and might happen when K is not sufficiently large. The search can fail if there is more than a single connected component, as not every vertex is accessible from any given starting point in the search index. We will discuss the implications of the choice of K in the experimental results.

Steps L and M also do not imply changes in the search index, as the existing connections are already the best options compared to the joined vertices. At the end of step M, all the existing vertices acted as the reference vertex. Therefore, one iteration of the refinement process was performed. The process is repeated until the convergence criteria are met. Step N shows the example graph after convergence. SWINN applies two tests to stop performing the refinement process:

1. The maximum number of iterations is reached ($\texttt{max}_{\text{iter}}$);

2. The total number of edge changes during an iteration of refinement is smaller than δKL, where δ is a convergence tolerance parameter.

We present the neighborhood refinement procedure in Algorithm 5. Next, we explain how the search is performed in SWINN.

Algorithm 5 – Graph refinement procedure.

Require:
$V_r \subseteq V$: a list of vertices to perform the graph refinement procedure;
$\texttt{max}_c$: the maximum number of candidates to consider in local neighborhood joins;
δ: convergence criterion;
$\texttt{max}_{\text{iter}}$: the maximum number of iterations allowed for graph refinement.

1: **procedure** SWINN-REFINEMENT(V_r, $\texttt{max}_c$, δ, $\texttt{max}_{\text{iter}}$)
2: **for** $i \in \{1, ..., \texttt{max}_{\text{iter}}\}$ **do**
3: **for** $v \in V_r$ **do**
4: Retrieve T_v and its corresponding binary flags B_v ▷ total neighborhood
5: Take a random sample, $S(T_v)$, of size $\texttt{max}_c$ out of T_v
6: **for** every $v' \in S(T_v) | B_v(v') = \texttt{true}$ **do**
7: Attempt to create edges between v' and vertices whose binary flags are `true`
8: Attempt to create edges between v' and vertices whose binary flags are `false`
9: **end for**
10: Finish the procedure if the total number of edge changes is smaller than $\delta K|V_r|$
11: **end for**
12: **end for**

7.4.2 Search

SWINN relies on a simple greedy search strategy to look for the NNs of a given query point. We start by explaining how search works in the 1-NN case. Afterward, we expand our discussion to include the cases where $k > 1$. Although we select a specific K value to build the index, during the search, the number of returned NNs (k) is only bound by L.

Figure 26 illustrates how SWINN carries out the search. In step A, we take the same graph obtained after the refinement performed in the last illustrative example. The query instance is signaled by x on the left side of the graphs. The search starts in step B by randomly choosing a starting seed vertex (the purple vertex). To reduce the search space, SWINN defines a search bound, given by the distance from the current best solution to the query. Step B represents the search bound by a circle centered in purple. The circle's radius corresponds to the distance from purple to the query point. Later, we will relax the definition of the search bound to decrease the chance of falling into local minima during the search.

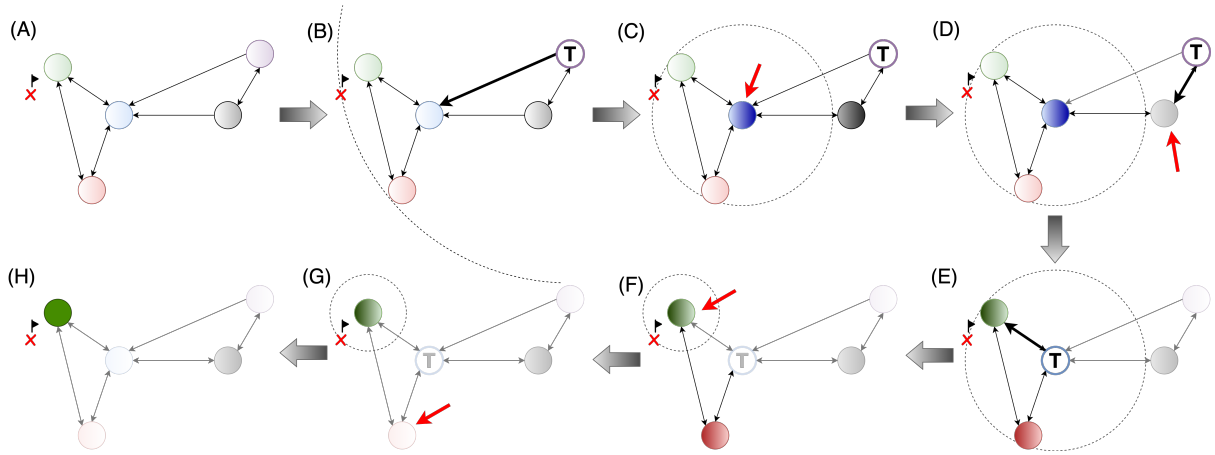


Figure 26 – Illustration of the search procedure on an already constructed search graph (step A). The search procedure starts on a random seed vertex (step B). The dashed circle represents the distance bound given by the current best solution. Candidates whose distance to the query point (represented by x) is larger than the distance bound are ignored.

The search procedure proceeds by selecting the best option among the current neighbors of the starting vertex. In our example, blue is the closest available vertex. In step C, we shift our focus to blue and update the distance bound. The vertex gray was not yet explored (step D), as it is also a neighbor of the seed vertex. However, the distance between gray and the query is larger than the updated distance bound. Therefore, gray is not further explored.

Next, in step E, blue becomes the target vertex, the current best solution. The search continues by exploring the neighbors of blue, which were not yet visited (green and red). The first selected neighbor is green. SWINN's vertices keep an unordered list of neighbors, so, in our example, green is first selected for illustration purposes. The red vertex could also come first if it were the first element in the list of total neighbors. As green is closer than blue to the query point, the distance bound is updated in step F. There is only one remaining vertex to explore

(step G), i.e., red. Nevertheless, the search ignores this vertex because the distance of red to the query is larger than the current distance bound.

The search process stops in step H when green is selected as the query result. It must be observed that, in this toy example, all the nodes were visited during the search. Thus, the search cost was asymptotically equivalent to a LS. However, in practice, the number of available vertices is much higher, and the combination of the greedy search strategy and the distance bound significantly reduces the search space and speedup graph traverse.

In the illustrated search process, the distance bound is the distance from the best vertex found so far to the query item. Nonetheless, by relying on this approach, the search process might end up in local minima. For example, suppose there was an even closer vertex to the query only accessible via red. The search would not reach this vertex by using the strict distance bound.

Alternatively, we can set a $\varepsilon \geq 0$ parameter to extend the distance bound, i.e., during the search, the distance bound will be defined as $(1+\varepsilon)d_b$, where d_b represents the distance between the current best solution and the query. Hence, vertices can still be explored, even though their distance to the query is (slightly) higher than the current best solution. We experiment with different values of ε in our experimental setup.

So far, we have only discussed how to search for the 1-NN element in the graph. We can generalize the search for the k-NN case using two binary heap data structures. The first heap, H_{min}, is a min-heap and keeps a pool of vertices whose neighborhood is yet to be explored. This heap first explores the vertices with the smallest distances to the query element. The second heap, H_{max}, is a max-heap and stores the search results. The head of H_{max} carries the farthest among the k-NNs.

During the search, elements are removed from the head of H_{min} and have their total neighborhood explored, as described in the search example illustrated in Figure 26. Each vertex in the total neighborhood of the head of H_{min} is a candidate to be added to both H_{min} and H_{max}. The distance bound, previously described, is calculated using the head element of H_{max}. A new vertex is added to H_{max} only if its distance to the query is smaller than the distance bound. In this case, the head element of H_{max} is removed, and the new candidate is added to both H_{min} and H_{max}. Note that H_{max} always carries at most k elements, whereas H_{min} is not bounded in length. The distance bound is also updated with the new head element in H_{max}. The search concludes when there are no remaining elements to explore in H_{min}. At this point, H_{max} has k elements partially sorted in the reverse order. These items can then be quickly sorted in increasing order according to their distance to the query and returned as the search result.

7.4.3 Vertex addition

We already have tools to refine the neighborhood of a random graph, and we can also perform search queries once the k-NN graph is created. Hence, we can add new elements to the

search graph using a simple strategy. When a new data point arrives, SWINN uses the existing graph to find the instances' K neighbors. SWINN then creates a new vertex to accommodate the new instance and adds direct edges to the K found vertices. Our proposal also updates the reverse neighborhood of the selected neighbors to account for the newly added vertex.

7.4.4 Element removal

In SWINN, instances are supposed to be constantly removed from and added to the graph in a sliding window. Hence, we need to ensure that the search index remains reliable, even though some of its nodes and edges will constantly change. As previously discussed, new elements can be added by using the current search graph to find the direct neighborhood of the new vertex. Vertex removal is more challenging than element addition.

A possible option for removal is to apply the needed changes in the index, i.e., remove nodes/edges, and then run the refinement procedure (subsection 7.4.1) using the whole graph. However, this action has a cost of $O((\texttt{max}_c)^2 L)$. Following this strategy is more costly than performing a linear scan (LS) in the data buffer. Another option is to perform graph refinements at predefined intervals. Even so, there are no guarantees that the graph will be reliable for NNS during the intervals between refinements. Besides, there is not even a guarantee that the graph will be reliable for performing searches after removing a single vertex and all its edges. We need to balance search index efficiency and reliability. For such, let us first define search reliability.

Lemma: a graph-based search index is reliable when there is only a single connected component, i.e., every node in the graph can be reached from any given starting point.

If this is not the case, the search is deemed to fail with a wrong choice of the starting point. Even if a seed vertex close to the query is selected, some vertices might become inaccessible. In SWINN, we perform local adjustments to the graph after each element removal. Therefore, only the local neighborhood of the removed vertex is updated.

We get the list of reverse and direct neighbors of the vertex to be removed, i.e., the oldest element in the sliding window. The oldest element is removed from the graph together with its edges. We then perform two filtering procedures in the previously retrieved neighborhood lists.

1. From the list of reverse vertices, only keep those with no direct neighbors;
2. From the list of direct neighbors, only keep those with no reverse neighbors.

On the one hand, we have a list of reverse neighbors without direct neighbors. On the other hand, we have vertices without reverse neighbors. Although none of the situations is inherently problematic, as long as there are alternative paths to reach one group starting from the other, we cannot directly guarantee that both groups are not separate, i.e., the two lists might

contain border vertices in two sub-graphs. We could verify if that is the case using a Minimum Spanning Tree algorithm (MST), such as Kruskal's (KRUSKAL, 1956), and verify whether a single MST or a forest thereof is obtained. Notwithstanding, Kruskal's algorithm has a cost of $O(E \log V)$, which in our case is $O(KL \log L)$, due to the nature of the search index. Running an MST building algorithm after almost every vertex removal is too costly. We propose an alternative solution to keep the graph reliable and computationally efficient.

For such, we first check if there is any intersection between the two filtered lists. If so, these vertices are isolated in the search graph. We fix this situation by re-adding them to the index. Next, for each node in the list of filtered reverse neighbors, we add new K neighbors using the graph search procedure. However, instead of randomly selecting a random search seed vertex among all those available, we randomly select a vertex from the second filtered list, i.e., the list of direct neighbors of the removed vertex. As a result, we might create connections between two separate sub-graphs. In case the second list is empty, we still add new neighbors to the members of the first list using random search seed vertices. After creating the new connections, SWINN applied the graph refinement procedure to the vertices from the two filtered lists.

7.4.5 Edge pruning

As vertices can have mutual edges and intersecting (undirected) neighborhoods, the resulting search index might contain multiple and, potentially, redundant paths between two given vertices. Even though some edges, at first glance, are worse than others, they still can be helpful during index searches, e.g., a given vertex's worst edge, w.r.t. distance, might provide the most direct path toward the search query. On the other hand, if paths are redundant, they will increase the search time and the memory footprint of the search index.

As discussed in subsection 7.4.2, SWINN relies on greedy search, as many of the preceding batch-based graph solutions. For this reason, at each step of the search, the total neighborhood of the current node must be expanded to select the potential best path to follow. At this point, the impact of having multiple paths between two vertices is twofold: (1) they can boost search accuracy by enabling faster hops towards the target vertex; (2) redundant paths increase the neighborhood exploration time with no direct benefits to the search accuracy.

The search graph can be pruned to reduce the number of edges and speed up the search. However, care must be taken when removing edges to avoid creating multiple connected components, i.e., sub-graphs. Due to the incremental nature of the underlying search process, we should avoid calculating distances between vertices to save computational resources. Hence, we propose a simple edge pruning procedure that accounts for calculated distances between vertices.

The main idea is akin to the refinement procedure employed to create the search graph. This time though, we want to "remove connections to neighbors of my neighbors if they are closer to my neighbor than they are from myself". SWINN creates a min-heap during the pruning

procedure to track the direct and reverse neighbors of a focus vertex v, ordered by their distance. The best neighbor is retrieved from the min-heap and added to a set of selected neighbors S. For each remaining candidate c in the neighborhood heap, SWINN first checks if an edge exists between c and one of the selected neighbors $s \in S$. When it is not the case, c is added to S. If an edge exists between c and an element $s \in S$, the farthest among the undirected edges (v,s) and (s,c) is discarded. Therefore, if we imagine a triangle formed by vertices v, s, and c, SWINN effectively removes the polygon's longest side. Following this strategy, v, s, and c are still mutually reachable even after removing the longest edge among them.

We illustrate how the pruning procedure works with Figure 27. This pruning procedure can be applied to all vertices in the search index, but in SWINN, we only apply it after performing (local) graph refinements. Figure 27 shows a hypothetical search graph where edge pruning is applied to the green vertex, which we refer to as the focus vertex. Step A shows the starting search index. At the beginning of the procedure, we create an empty list of selected neighbors. All the existing neighbors of the focus vertex are candidates for either removal from or addition to the list of selected neighbors. Only the selected neighbors will remain at the end of the pruning procedure. We compare each neighbor of green against the list of selected neighbors accordingly to their distance. If a candidate vertex is also a neighbor of one vertex in the list of selected vertices, only the best edge among two compared vertices is kept.

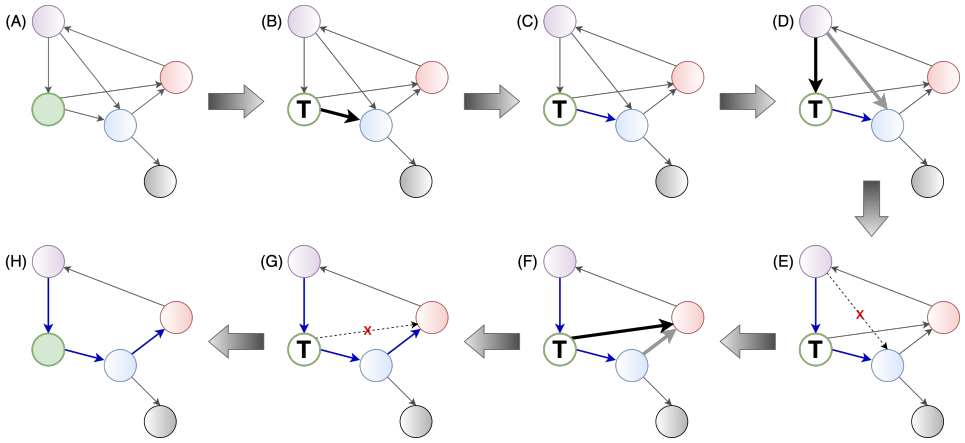


Figure 27 – Illustration of the edge pruning capabilities starting from the green vertex. Starting from the top left to the bottom right: each edge of green is selected, and the smallest are kept. Edges (green, red) and (purple, blue) are removed by the procedure. Note that two hops will be needed to reach red starting from green after removing the edges. Similar cases happen for the other affected vertices.

In step B of Figure 27, the best available edge, blue, is selected for analysis (highlighted in **bold**). As there are no selected neighbors yet, blue is added to the list of selected neighbors (step C). Next, in step D, the second best edge (purple, green) is analyzed, and the list of selected nodes is checked for possible redundant paths. In fact, purple is also a neighbor of blue, which was previously selected. As the edge between purple and green is shorter than that between purple and blue, the latter edge is removed (step E). In step F, the third and last neighbor of green

is selected (red) for analysis. Indeed, red is also a neighbor of blue. Again, the shortest edge among the two compared vertices is kept, leading to the removal of red as a neighbor of green (step G). The procedure stops at step H, as there are no more edges to explore from the focus vertex.

Even though the previous procedure guarantees that vertices are accessible and the index is not split into multiple connected components, care must be taken to remove edges. Due to the usage of a greedy search algorithm, long edges might help to avoid local minima. Hence, neither keeping all the edges nor removing all possible redundant paths might be optimal depending on the considered performance measure. For this reason, we add the parameter $\texttt{prune}_{\text{prob}}$ to control the probability of removing a redundant edge. Therefore, the pruning procedure works as previously described, with the difference that each time a redundant edge is found, there is a probability, $1 - \texttt{prune}_{\text{prob}}$, of not removing it from the graph. The value of $\texttt{prune}_{\text{prob}}$ is a user-given parameter. When $\texttt{prune}_{\text{prob}} = 0$, no edges are removed, whereas when $\texttt{prune}_{\text{prob}} = 1$, all possibly redundant edges are going to be removed.

7.4.6 The complete SWINN functioning

In this section, we show how each previously presented element of SWINN works in combination with the others. The complete SWINN updating procedure is presented in Algorithm 6.

7.5 Experimental setup on simulated data

In this section, we describe the experimental setup used in our experiments, including the data, the evaluation measures, the baseline, and the settings used in SWINN. SWINN was implemented in Python and will be incorporated into River (MONTIEL *et al.*, 2021) for ease of public use and access. All the reported experiments were performed sequentially in a CentOS machine with 2 Intel Xeon E5-2667v4 processors with eight cores running at 3.2 GHz and 512GB of DDR3 RAM.

7.5.1 Baseline, data and evaluation metrics

For the baseline, we compared the performance of SWINN to a linear scan (LS) of the data buffer. LS is also sometimes referred to as an exhaustive search. First, we compared our proposal against the baseline by using synthetic data. Our goal was to understand better how the hyperparameters involved in the design of SWINN affect its predictive performance.

In our experiments, we generated 10 uniformly sampled features and 50 000 instances for each case study. This experimental setup did not consider concept drift, as sliding window-based learning models are naturally adaptive by keeping only the most recent data. The evaluation

Algorithm 6 – The complete SWINN update function.

Require:
D: the data stream;
K: the number of direct neighbors each vertex must have;
$\texttt{max}_c$: the maximum number of candidates to consider in local neighborhood joins;
$\texttt{prune}_{\text{prob}}$: the probability of removing a long edge;

function SWINN-UPDATE(D, K, $\texttt{max}_c$, $\texttt{prune}_{\text{prob}}$)
  Let Q be a queue with at most L elements and G be an empty graph
  while D has instances **do**
    $\texttt{item} \leftarrow \texttt{next}(D)$
    Let v_{new} be a new vertex containing `item` and no edges
    if G is empty and $|Q| < L$ **then**
      Add v_{new} to Q
      if $|Q|$ is equal to the warm-up period **then**
        Create a random graph $G = (V, E)$ using all elements in Q
        Refine G using all the vertices (subsection 7.4.1)
        Skip to the next instance
      end if
    end if
    if $|Q| = L$ **then**
      Remove the oldest vertex, v_{old}, from the graph
      Fix the previous neighborhood, $T_{v_{\text{old}}}$, of v_{old} (subsection 7.4.4)
      Apply the graph refinement procedure to the vertices in $T_{v_{\text{old}}}$
      for $v \in T_{v_{\text{old}}}$ **do**
        if $|T_v| > \texttt{max}_c$ **then**
          Prune redundant edges of v with probability $\texttt{prune}_{\text{prob}}$
        end if
      end for
    end if
    Find the K nearest neighbors of v_{new} using G (subsection 7.4.2)
    Add v_{new} along with edges to its K nearest neighbors to G
    Add v_{new} to Q
  end while
  return G
end function

strategy followed a test-than-train approach (BLUM; KALAI; LANGFORD, 1999), i.e., the popular prequential evaluation approach used in data stream learning (GAMA; SEBASTIAO; RODRIGUES, 2009). In this strategy, for every incoming instance, we first retrieve the position of its k-NN in the sliding window and then add the new example to the buffer.

The query results of the LS baseline are used as the ground-truth values to calculate the Search Recall (SR) values, which vary between $[0, 1]$, where 1 is the best possible value. Thus, SR represents the mean percent of the true NNs that SWINN can retrieve. As LS checks every possible example in the data buffer, its SR is always 1. Ideally, we want to obtain values as close to 1 as possible with SWINN while being faster than LS performing search queries. Hence, we

measure the running time of both LS and SWINN in seconds, considering element insertion, search query, and the total run time. We also measure the memory footprint of the compared algorithms for NNS.

7.5.2 Graph configurations

Table 22 presents the list of hyperparameters assessed for SWINN. We separate the hyperparameters into two categories: index building and search. We left the hyperparameters values of the two convergence criteria fixed during our experiments, following guidelines defined in the literature (DONG; MOSES; LI, 2011; BRATIĆ *et al.*, 2019). Moreover, we set SWINN to use the first 100 instances in the stream to warm-up. We discuss the impact of changing the values of the remaining hyperparameters in the experiments' discussion section. The values of L (window length) were applied to both SWINN and the LS baseline.

Table 22 – List of SWINN hyperparameters considered in our experimental setup. The choice of L and k also applies to the Linear Scan baseline.

Stage	Name	Description	Value(s)
Build	δ	Convergence criterion	0.001
	$\texttt{max}_{\text{iter}}$	Maximum number of iterations for graph refinement	10
	L	Sliding window length	$\{100, 250, 500, 1000, 5000\}$
	K	Number of neighbors during graph construction	$\{5, 10, 20, 30\}$
	$\texttt{max}_c$	Maximum number of candidates to consider during local neighborhood joins	$\{20, 30, 50, 100\}$
	$\texttt{prune}_{\text{prob}}$	Probability of pruning a long edge	$\{0.0, 0.3, 0.5, 0.7, 1.0\}$
Search	ε	Tolerance hyperparameter for the distance bound	$\{0.0, 0.1, 0.5, 1.0\}$
	k	Number of neighbors to retrieve during search queries	$\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$

7.6 Results and discussion

In this section, we evaluate the performance of SWINN in the synthetic data described in subsection 7.5.1. We start by analyzing how each hyperparameter affects SWINN's performance and later compare SWINN against the LS of the data buffer. In section 7.7, we apply our findings in classification and regression case studies using synthetic and real-world data.

Multiple aspects have an impact on the performance of the search index, e.g., it is widely known that distance-based methods suffer the so-called "curse of dimensionality". The NN-Descent authors mention in the original paper that their method works better for data with less than 20 features (DONG; MOSES; LI, 2011). Besides, the efficacy of a search index is closely related to the characteristics of the data. In this section, we limit our analysis to a controlled experimental setup to better understand each aspect of SWINN and its impact on predictive performance.

External factors, such as data dimension and hubness (DONG; MOSES; LI, 2011; BRATIĆ *et al.*, 2018; BRATIĆ *et al.*, 2019), remain open questions and should be addressed in the future. However, as we operate in a sliding window environment, possible sources of

search instability are transient. For example, it is known that nodes with an increased number of total neighbors affect NN-Descent in a negative way (BRATIĆ *et al.*, 2018; BRATIĆ *et al.*, 2019). In SWINN, as more data are observed, these nodes are eventually removed from the graph. Therefore, some challenges faced by batch applications might not affect SWINN to the same extent.

We organize our discussion by formulating the following research questions, which will be addressed in the next subsections:

Q1: Is SWINN effective regardless of the sliding window size?

Q2: Does the number of requested nearest neighbors impact the search recall during graph search?

Q3: How ε value impacts search recall and running time during search queries?

Q4: What is the impact of the chosen number of neighbors used to build SWINN's graph on search recall and running time?

Q5: What is the impact of neighborhood sampling when performing local joins during graph refinement?

Q6: What is the impact of edge pruning on the overall search recall and memory and time costs?

7.6.1 Sliding window size

We ran every hyperparameter combination described in subsection 7.5.2 and checked the effectiveness of SWINN accordingly to the running time and SR. We report our findings in Table 23 accounting for hyperparameter value combinations that were faster than a LS while also delivering SRs larger than 0.9. When considering only time, no gains were observed for $L = 100$. In fact, in our experiments, we set the warm-start period to 100 instances, as reported in subsection 7.5.2. As shown in the table, gains in processing time are only noticeable starting at $L = 250$. SWINN time efficiency becomes more apparent as the window length increases, and our proposal was always faster than an LS when $L = 5000$.

Table 23 – Overall performance of SWINN compared to a linear scan of the data window, considering time and search recall.

Window length	**100**	**250**	**500**	**1000**	**5000**
$\text{Time}_{\text{SWINN}} < \text{Time}_{\text{LS}}$	0.00%	21.31%	48.72%	77.56%	100.00%
$\text{SR}_{\text{SWINN}} > 0.9$	100.00%	65.62%	61.87%	55.37%	46.78%
Both	0.00%	0.00%	13.72%	32.97%	46.78%

Still, considering $L = 250$ and SR, no hyperparameter value combination delivers SR values higher than 0.9 while also being faster than a LS. We only start to observe hyperparameter value combinations able to be faster than LS and with SR > 0.9 when using $L = 500$. Nonetheless, the amount of hyperparameter value combinations able to deliver the combined gains are minimal (13.72%). Therefore, answering the research question **Q1**, we conclude that SWINN is better suited for windows with more than 500 instances. Hence, by setting the warm-up period to 500, one can achieve a good compromise in performance. In the following sections, we investigate the effect of the other hyperparameters of our technique and check how they impact SWINN's performance. We will, from here onward, limit our analysis to $L = 1000$ and $L = 5000$.

7.6.2 The impact of ε on graph search

The greedy search used in SWINN might become stuck in local minima. To deal with this limitation, we include the ε hyperparameter, which allows SWINN to explore vertices slightly worse than the available nearest vertex at each search step. Nonetheless, overextending the number of explored vertices during the search has a toll on the running time. Hence, a balance must be achieved by setting a proper ε value.

We analyze how the ε hyperparameter influences the running time and SR of SWINN in general. For such, we select the most unconstrained version of SWINN among our experimental setup, employing $K = 30$ neighbors to build the search index, using $\texttt{max}_c = 100$, and disabling edge-pruning. We select this combination to minimize other parameters' effects on the SR. In particular, the choice of the highest evaluated K value guarantees a single connected component in the search index and multiple redundant paths between vertices.

In Figure 28, we present our analysis considering queries for the 1-st until the 10-th nearest neighbor. When analyzing the left side of the figure (first column), the choice of $\varepsilon = 0$ has a negative impact on SR, especially when $k = 1$. SWINN produces SR values close to 0 when searching the 1-NN, regardless of the window length. There is a significant jump in SR for every selection of $\varepsilon > 0$. Still, we observe that the higher the value of k, the higher the SR, a finding that answers the research question **Q2**. This behavior is related to how the search is performed in the graph. The distance bound, described in subsection 7.4.2, is defined by the worst neighbor among the current best candidates. Thus, when $k = 10$, the 10-th nearest neighbor at any given point of the search will define the threshold to either explore or ignore the neighborhood of a candidate node. When $k = 1$, only a candidate will be kept as a solution, and the inclusion criterion will become more restrictive. In these cases, increasing the ε value is mandatory.

Overall, $\varepsilon = 0.1$ provides the best balance between SR and running time, especially as the number of queried neighbors increases. Nonetheless, other values of ε might be selected depending on the primary goal. When searching for an increased number of neighbors, the choice of ε has little impact on SR, although it saves computation time. These observations answer the research question **Q3**. From this point onward, we select $\varepsilon = 0.1$ to analyze the impact of K,

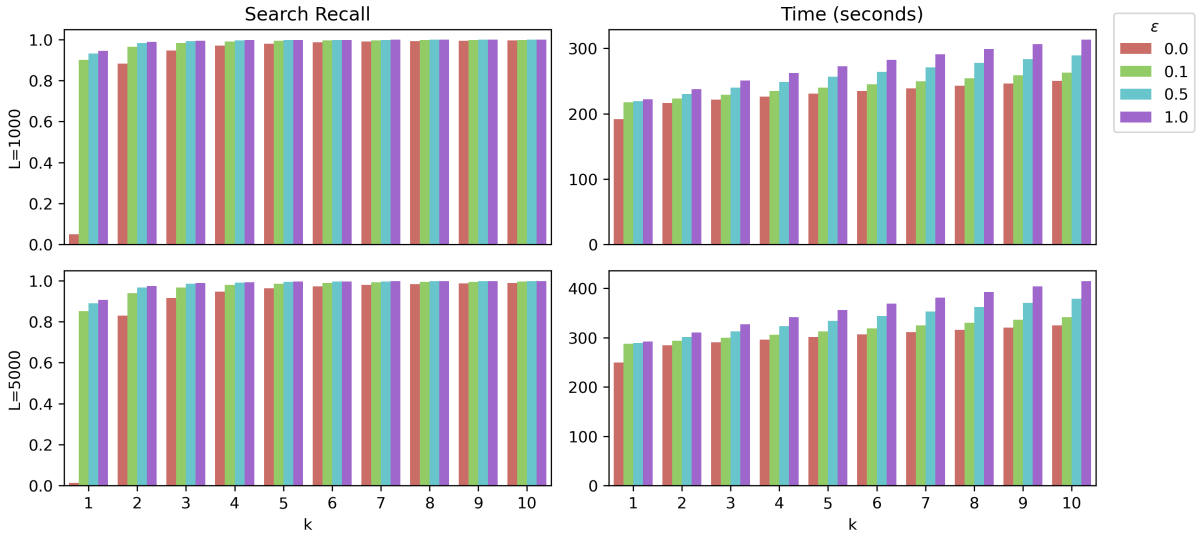


Figure 28 – The impact of ε on the total search recall (left) and search time (right). From top to bottom: window lengths of 500, 1000, and 5000 instances. Each color represents a ε value, and each group of bars represents the number of neighbors searched in each query.

$\texttt{max}_c$, and $\texttt{prune}_{\text{prob}}$ in SWINN.

7.6.3 The impact of the value of K on the search index

So far, we have only analyzed factors external to the search index building. These factors directly impact the search graph's performance but do not dictate how it is built. We now analyze how the choice of the value of K impacts the SR, the running time, the memory footprint, and the number of connected components in the search graph. We present our results in Figure 29 considering $L = 1000$. Similar results were observed with $L = 5000$ and can be checked in the Appendix.

To answer the research question **Q4**, we start by considering the running time measurements, depicted in the first column of Figure 29. As expected, the running time to add elements to the graph and perform search queries increases as the value of K increases. Nonetheless, we also must consider how the choice of the value of K impacts the other performance measurements. Looking at the top-right chart of Figure 29, we perceive that $K = 5$ yields significantly smaller values of SR in comparison with the other values of K. The bottom-right chart gives us the possible reason. Every value of K, except for $K = 5$, results in a graph with a single connected component. On the other hand, $K = 5$ generates more than one sub-graph on multiple occasions. Hence, SR results are expected to degrade as not every vertex will be reachable. For $K > 5$, a single connected component is always obtained. To answer the research question **Q4**, in general, the higher the K, the more accurate SWINN becomes, at the cost of increased computational resource usage.

Our goal is to find the best balance between time, SR, and search reliability. We believe that $K = 20$ represents a good compromise between all the metrics and will be used from this

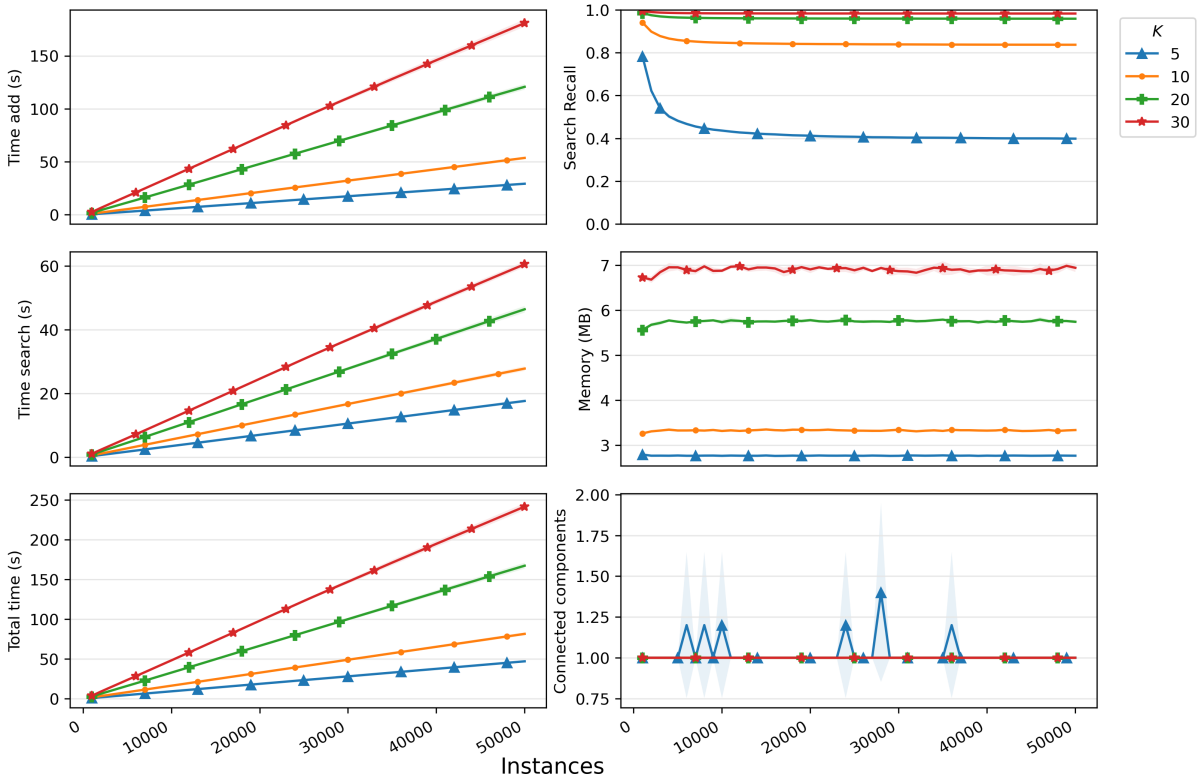


Figure 29 – The effect of increasing the value of K in SWINN, when $L = 1000$.

point onward. The downside of this choice, depicted in the middle chart on the right side of Figure 29, is the increased use of memory to store the graph. As expected, the memory footprint of SWINN using $K = 20$ is smaller than the version with $K = 30$. Still, it is more costly than performing an LS in the data buffer. SWINN spends additional memory resources to keep all the vertices and edges in addition to all the instances in the buffer. However, our primary focus is to improve the running time performance and keep competitive SR values.

7.6.4 The impact of neighborhood sampling during local joins

We also evaluate the impact of $\texttt{max}_c$ in the overall performance of SWINN. We used $K = 20$, as previously mentioned, and disabled edge-pruning. We present the results for $L = 5000$ in Figure 30, while the results for $L = 1000$ are included in the Appendix.

As it can be observed, the running time differences between different values of $\texttt{max}_c$ are negligible. In fact, versions that bound the number of neighbors in local joins to 20 and 30 are slightly slower than the less restrictive versions of SWINN. This is due to the cost of performing vertex sampling during the local joins. These differences are also hard to notice when using $L = 1000$.

To answer the research question **Q5**, we believe that limiting the number of vertices participating in local joins is useful when the total neighborhood is high, i.e., when the value of K is high, and vertices have an increased number of reverse neighbors. The user controls the

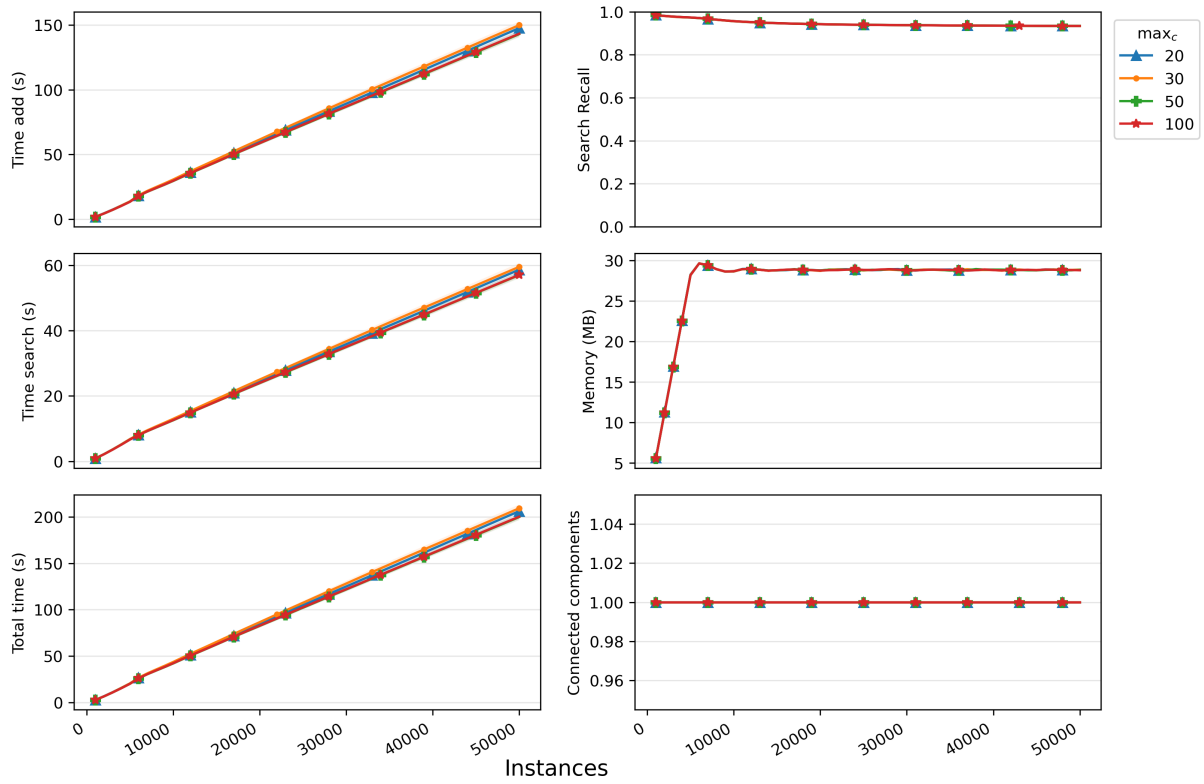


Figure 30 – The effect of increasing the value of $\texttt{max}_c$ in SWINN, when $L = 5000$.

first factor, while the second is data-dependent. As a compromise, we select $\texttt{max}_c = 50$ to have a possible balanced solution, regardless of the window length.

7.6.5 The impact of edge pruning

Finally, we evaluate the impact of edge pruning in SWINN, illustrating the effect of increasing the chance of removing possibly redundant edges in Figure 31. For such, we selected $L = 5000$, but similar observations can be made to $L = 1000$.

It is difficult to observe changes in the running time and SR in this figure. The number of connected components remains unchanged, as desired. We cannot perceive significant variations in the memory footprint of SWINN. Unlike the batch graph-based search index, the edges in SWINN have a transient characteristic. Edge pruning could be helpful in cases where the value of K is higher than the values we evaluated in our experimental setup. Even in these cases, every node is eventually removed, even the so-called hubs in related literature (BRATIĆ *et al.*, 2018; BRATIĆ *et al.*, 2019). To answer the research question **Q6**, the benefits and drawbacks of edge pruning are shadowed by the naturally evolving characteristics of sliding windows-based nearest neighbor search. Therefore, we adopt $\texttt{prune}_{\text{prob}} = 0$ in the follow-up experiments, i.e., no edge pruning.

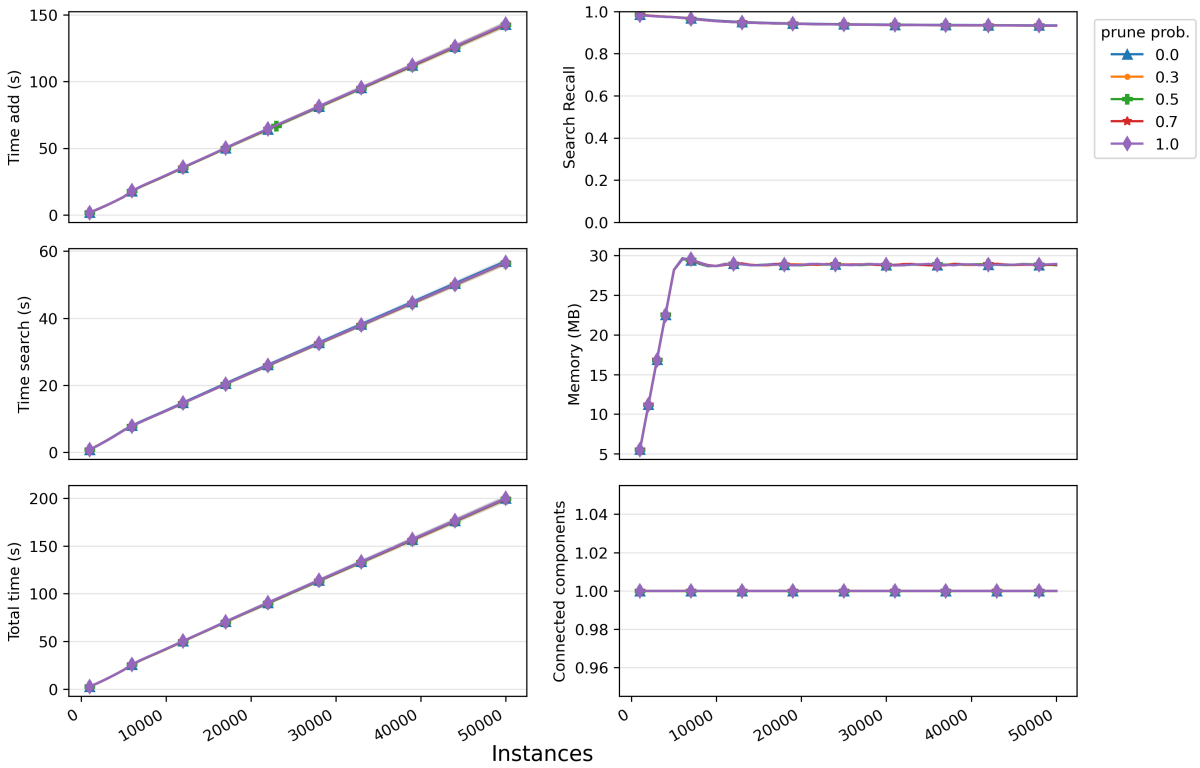


Figure 31 – The effect of increasing the value of $\texttt{prune}_{\text{prob}}$ in SWINN when $L = 5000$.

7.6.6 Comparing SWINN against a linear scan of the data

We also compare the performance of SWINN against a complete LS of the data buffer using the hyperparameter values combination obtained in the last section. In other words, we use SWINN with $K = 20$, $\texttt{max}_c = 50$, and $\texttt{prune}_{\text{prob}} = 0$. We present the obtained results in Table 24, considering windows of 1000 and 5000 instances. It is no surprise that the memory usage of SWINN is higher than LS'. This occurs because SWINN uses additional memory resources to store the search index, whereas LS only stores the instances per se. Notwithstanding, our primary focus is the running time, and even for large data windows, the amount of memory used by SWINN is easily manageable in traditional data stream setups.

Regarding SR, SWINN cannot match LS, regardless of the L value. This comes from relying on a greedy search strategy to perform queries in the search graph. Hence, the search is prone to be stuck in local minima, as it is primarily noticed when searching and using $k = 1$. As discussed in subsection 7.6.2, ε can be used to enhance the greedy search and significantly improve the SR when the value of k is small. Still, when $k \leq 2$, the choice of $\varepsilon = 0.1$ is not enough to make the SR values approach those obtained when $k > 2$, usually over 0.97. As an alternative, the user could increase ε further when searching for a small number of nearest neighbors. Nonetheless, in the future, we intend to search for alternatives to the greedy search currently used in SWINN and its batch counterpart.

Running time was the characteristic where SWINN truly shined. In Table 24, we highlight

Table 24 – Mean Search Recall, memory usage, and running time of LS and SWINN when considering $L = 1000$ and $L = 5000$. We indicate the mean memory usage of LS and SWINN alongside the window length. The number around parenthesis in the time measurements indicates how much faster SWINN performed in comparison with LS.

k	$L = 1000$ (LS: 1.22MB, SWINN: 5.76MB)				$L = 5000$ (LS: 6.11MB, SWINN: 28.86MB)			
	SR		Time		SR		Time	
	LS	SWINN	LS	SWINN	LS	SWINN	LS	SWINN
1	**1.00 ± 0.00**	0.81 ± 0.00	218.57 ± 13.87	**146.63 ± 3.45** (×1.49)	**1.00 ± 0.00**	0.75 ± 0.00	1370.13 ± 18.67	**179.00 ± 3.72** (×7.65)
2	**1.00 ± 0.00**	0.92 ± 0.00	218.77 ± 13.98	**150.86 ± 3.48** (×1.45)	**1.00 ± 0.00**	0.87 ± 0.00	1369.02 ± 20.62	**184.05 ± 3.31** (×7.44)
3	**1.00 ± 0.00**	0.95 ± 0.00	218.17 ± 13.85	**155.14 ± 3.59** (×1.41)	**1.00 ± 0.00**	0.92 ± 0.00	1367.35 ± 19.23	**188.87 ± 3.19** (×7.24)
4	**1.00 ± 0.00**	0.97 ± 0.00	218.52 ± 13.78	**159.22 ± 3.69** (×1.37)	**1.00 ± 0.00**	0.95 ± 0.00	1348.04 ± 19.71	**193.50 ± 3.19** (×6.97)
5	**1.00 ± 0.00**	0.98 ± 0.00	218.99 ± 13.30	**163.22 ± 3.82** (×1.34)	**1.00 ± 0.00**	0.96 ± 0.00	1359.93 ± 19.66	**198.09 ± 3.22** (×6.87)
6	**1.00 ± 0.00**	0.98 ± 0.00	218.55 ± 14.49	**167.07 ± 3.88** (×1.31)	**1.00 ± 0.00**	0.97 ± 0.00	1350.87 ± 18.44	**202.61 ± 3.24** (×6.67)
7	**1.00 ± 0.00**	0.99 ± 0.00	220.07 ± 13.89	**170.83 ± 3.97** (×1.29)	**1.00 ± 0.00**	0.97 ± 0.00	1368.97 ± 21.69	**207.02 ± 3.29** (×6.61)
8	**1.00 ± 0.00**	0.99 ± 0.00	218.33 ± 13.90	**174.36 ± 4.05** (×1.25)	**1.00 ± 0.00**	0.98 ± 0.00	1352.17 ± 19.21	**211.29 ± 3.40** (×6.40)
9	**1.00 ± 0.00**	0.99 ± 0.00	219.44 ± 14.60	**177.91 ± 4.08** (×1.23)	**1.00 ± 0.00**	0.98 ± 0.00	1366.96 ± 20.72	**215.59 ± 3.37** (×6.34)
10	**1.00 ± 0.00**	0.99 ± 0.00	218.58 ± 13.58	**181.19 ± 4.24** (×1.21)	**1.00 ± 0.00**	0.98 ± 0.00	1355.69 ± 20.67	**219.78 ± 3.47** (×6.17)

the number of times SWINN was faster than LS inside parenthesis. When $L = 1000$, SWINN was always faster than LS, though sometimes the speedup of our proposal was at most 20%. However, when we move to windows of 5000 instances, SWINN is generally at least 6 times faster than the LS. We believe SWINN is a solid choice to perform k-NN for larger window sizes, which would be impractical using LS.

7.7 Case studies

We investigated two case studies to assess how SWINN performs when applied to supervised ML tasks. Unsupervised learning is also viable, e.g., anomaly detection, but we wanted straightforward ways of comparing predictive performance. Our idea is to define a limited time to allow different classification and regression algorithms to run and verify their effectiveness in predictive performance and data processing throughput.

In these studies, we used data generators for classification and regression tasks. Data generators can produce synthetic instances indefinitely and allow the user to control the data characteristics. We allowed each compared algorithm to run for one hour in the same machine[1], in which each datum from the data generators was processed sequentially, using the prequential strategy. We also used similar classification and regression algorithms in both cases. All the algorithms are available in River (MONTIEL *et al.*, 2021) and were instantiated with their default hyperparameter settings, as implemented in River. For the k-NN models, we set $k = 5$ in all the cases, which is also the default in River. As for SWINN, we selected the same set of hyperparameters reported in section 7.6. Table 25 presents a list of the compared algorithms used in the classification and regression tasks.

Next, we provide more details about the experiments carried out and the results obtained for each case study.

[1]We used the same computer described in our experimental setup.

Table 25 – Compared **C**lassification and **R**egression algorithms and their acronyms.

Acronym	Meaning	Reference
LR	Linear Regression (R) or Logistic Regression (C)	-
HAT	Hoeffding Adaptive Tree	(BIFET; GAVALDÀ, 2009)
ARF	Adaptive Random Forest	(GOMES *et al.*, 2017; GOMES *et al.*, 2018)
LS(L)	k-NN using LS and a window with L instances	-
SWINN(L)	k-NN using SWINN and a window with L instances	-

7.7.1 Regression

We start by discussing a regression task. For such, we have chosen the Friedman Drift data generator (IKONOMOVSKA; GAMA; DŽEROSKI, 2011b), which is available in the River library. We selected the variant with Local and Expanding Abrupt (LEA) concept drifts, denoted by Friedman(LEA). This variant of the data generator implements three concept drifts affecting the feature space locally. The affected portions expand after each drift. Hence, the last concept drift is the most pronounced. We set the drifts to occur after 250 000, 450 000, and 1 500 000 instances.

The selected performance metrics were the Root Mean Squared Error (RMSE) and the Coefficient of determination (R^2), due to their popularity and complementary nature. For the former metric, the smaller its value, the better, whereas the opposite happens with the latter metric. The best possible value of R^2 is 1, when the regression model perfectly captures the underlying regression patterns. When R^2 equals 0, there is no correlation between the regressor output and the ground truth.

We added an incremental standard scaling procedure at the beginning of the processing pipeline of each regressor. The default behavior of Hoeffding Tree-based regressors in River is to build model trees, i.e., decision trees whose leaves carry LR models to provide predictions. That is the case with ARF and HAT regressors. The LR and NNS models work better when all the features are on the same scale. Therefore, all regressors had the input data scaled before processing each datum.

We present the results of the regression case study in Figure 32. We adopted the logarithmic scale for the x-axis due to the large gap in the number of processed instances by the different algorithms. LR and HAT were the fastest algorithms due to their simplicity and efficiency. These were the only two regressors that reached the last concept drift point. Indeed, we can perceive an increase in the predictive error of LR and HAT past 10^6 instances due to concept drift. However, only HAT could start reacting to the drift and gradually reduce the predictive error. LR is not equipped to deal with concept drifts, and its RMSE presents a pronounced increase. The remaining regressors did not reach the last concept drift point.

Table 26 details the differences between the compared regression models. In the table, we can see the large gap between the number of processed instances by LR and the remaining regressors. On the other hand, LR yielded the worst values of RMSE and R^2 among the com-

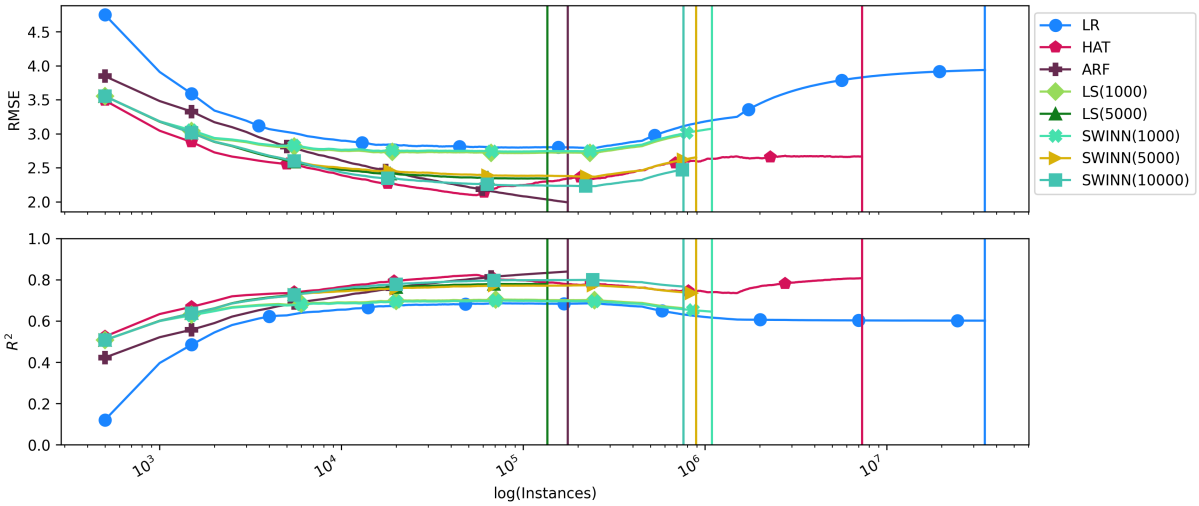


Figure 32 – Results of different regressors when processing an instance of the Friedman(LEA) data generator for one hour. The x-axis is in a logarithmic scale due to the large differences between the compared models. Vertical lines denote the maximum number of instances each model could process during the allowed time.

pared regressors. HAT was around nine times faster than the fastest NNS-based regressor, i.e., SWINN(1000). Tree-based models had the edge in predictive performance, being ARF the most accurate algorithm. LR, HAT, LS(1000), and all the SWINN variants processed more instances than ARF during the allowed time.

By increasing the L parameter, SWINN-based k-NN regressors could surpass LS(1000) in predictive performance and still process a considerable number of instances. Among the k-NN models, LS(5000) was the most accurate, although also the slowest model. In all the cases, SWINN was over four times faster than LS(5000). The SWINN-based k-NN regressors obtained RMSE and R^2 values close to those obtained by the LS variants. By tweaking the values of ε and K, the predictive performance gap between LS and SWINN could be reduced, while the speed differences could be increased even further. Notwithstanding, we acknowledge that more efficient and effective graph query strategies must be applied to the search index. The greedy nature of the graph search, allied with a simple strategy to select the query starting point, might cause the search to be stuck in local minima. Still, SWINN has shown potential to be an efficient strategy for performing NNS in regression tasks.

7.7.2 Classification

In the classification case, we relied on the data generator, denoted Random Radial Basis Function (RBF), with gradual concept drifts (RBF-GD). We set up RBF-GD to create 20 features, 2 classes, and 50 micro-clusters from which 10 slowly shift (the change speed was set to 0.01).

Feature scale does not impact the classification version of HAT and ARF. Hence, we only applied feature scaling to LR and the NNS-based classifiers. We know that keeping an incremental feature standardization pipeline component and scaling all the features incurs extra

Table 26 – Regression case study using the Friedman(LEA) dataset. The reported RMSE and R^2 values correspond to the measurements after processing incoming data for one hour.

Algorithm	Instances processed	LS(5000) speed up	RMSE	R^2
LR	45 957 500	258.92×	3.94	0.60
HAT	9 762 500	55.00×	2.67	0.81
ARF	231 500	1.30×	1.99	0.84
LS(1000)	949 500	5.35×	2.98	0.66
LS(5000)	177 500	1.00×	2.34	0.78
SWINN(1000)	1 186 500	6.68×	3.07	0.65
SWINN(5000)	871 000	4.91×	2.65	0.73
SWINN(10000)	764 000	4.30×	2.47	0.77

running time costs. Nonetheless, that is the nature of linear and NNS models: feature scale matters. As we aimed for a realistic comparison of the algorithms, we opted to evaluate the classifiers in the same way they would be applied in real-world scenarios.

We present the evolving performance curves for Accuracy and the F1 score in Figure 33. Once again, LR and HAT were the fastest approaches, as expected. The ARF classifier, differently from its regression counterpart, was able to surpass the k-NN models in terms of the number of processed instances. Hoeffding Trees for classification store counters as split-enabling statistics and rely on highly efficient approximation techniques (PFAHRINGER; HOLMES; KIRKBY, 2008) to evaluate split candidates. Regression trees keep variance estimators as split-enabling statistics and rely on more costly split evaluation procedures due to the continuous nature of the labels. For that reason, the classification version of ARF is faster than the ARF regressor. The same holds when comparing the HAT classifier and regressor.

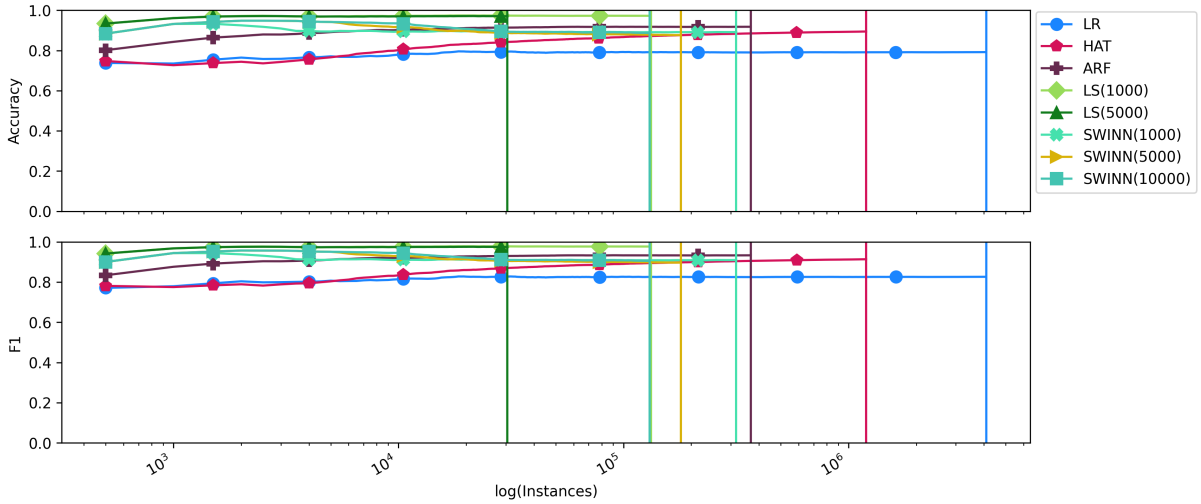


Figure 33 – Results of different classifiers when processing an instance of the RBF-GD for one hour. The x-axis is a logarithmic scale due to the large differences between the compared models. Vertical lines denote the maximum number of instances each model was able to process during the allowed time.

Looking at the final results in detail, as reported in Table 27, we realize that the per-

formance differences between SWINN and LS were more apparent in classification tasks. We argue that the number of selected features impacted the predictive performance. Also, due to the discrete nature of the labels, the approximation nature of SWINN's NNS might have a higher impact on the final performance. This observation comes from the fact that the most common label among the returned neighbors is the final answer. In regression, the average label value is the output. Hence, the influence of false positives in a search query might be more pronounced in classification tasks.

With the increase in dimensions, hubness (BRATIĆ *et al.*, 2019) might also impact the graph structure. The curse of dimensionality might also influence the greedy search applied in SWINN. The influence of the number of features on performance has been addressed by NN-Descent authors (DONG; MOSES; LI, 2011) and subsequent works (BRATIĆ *et al.*, 2018; BRATIĆ *et al.*, 2019). It remains an open issue to further study in online ML scenarios. Increasing the value of ε may increase SWINN's recall in these cases at the cost of slightly higher running time. Even in such settings, SWINN ought to be faster than performing an LS of the data buffer when the value of L is high, as the final results reported in Table 27 hint at. Even when using $L = 10000$, our proposal processed almost the same number of instances as the LS(1000) The differences in predictive performance between LS and SWINN give us more pieces of evidence to support focusing on graph search as the next step.

Table 27 – The classification case study's results using an instance of the Random RBF with gradual drifts dataset. The reported accuracy and F1 values correspond to the measurements after processing incoming data for one hour.

Algorithm	Instances processed	LS(5000) speed up	Accuracy	F1
LR	4 090 500	134.11×	0.79	0.83
HAT	1 198 000	39.28×	0.89	0.91
ARF	368 000	12.07×	0.92	0.93
LS(1000)	132 500	4.34×	0.97	0.98
LS(5000)	30 500	1.00×	0.97	0.98
SWINN(1000)	317 000	10.39×	0.89	0.91
SWINN(5000)	180 000	5.90×	0.88	0.90
SWINN(10000)	130 500	4.28×	0.89	0.91

7.8 Final considerations

In this paper, we proposed Sliding Window-based Incremental Nearest Neighbors (SWINN), an online and graph-based nearest neighbor search (NNS) algorithm. SWINN is primarily meant to work in a sliding window regimen, where new instances arrive, and the old ones are discarded. Nonetheless, any vertex of SWINN's search graph can be removed, and new vertices can be added. Therefore, other types of data ingestion can also be explored.

Our experiments show that SWINN effectively deals with online NNS when the sliding window size is sufficiently large ($L \geq 1000$). SWINN is significantly faster than a complete linear

scan (LS) of the sliding window while keeping competitive search recall to the LS approach. Furthermore, SWINN can deal with arbitrary distance metrics and copes with concept drift similarly to the LS strategy. Our proposal is also effective when applied to online ML tasks, from which we give classification and regression examples. We compare SWINN-based k-NN models against popular online classification and regression algorithms.

In future work, we intend to further explore search strategies in SWINN. SWINN relies on a simple bounded greedy search to perform graph searches. Moreover, search initialization is performed by selecting a single random vertex as the starting point. Both the search strategy and the search initialization can be further improved. An example would be using a multi-vertex search start and exploring additional search heuristics to perform graph traversal.

Additionally, applying SWINN as the building block of more complex k-NN-based solutions could be an exciting venue to explore. For instance, our proposal could be used in multi-memory solutions, where two levels of data buffering are applied, namely, short and long-term memories. The short-term memory holds the most recent data items in a sliding window fashion, while a larger window, possibly populated via reservoir sampling, carries examples of old concepts. Decisions are taken by performing NNS in both windows and combining the answers found in each memory level. Last, we intend to explore other distance metrics and investigate how the choice of the metric impacts the performance of online ML tasks.

Acknowledgments

This research was supported by the São Paulo Research foundation, grant #2021/10488-7. The experiments were carried out using the computational resources from the Center for Mathematical Sciences Applied to Industry (CeMEAI) funded by FAPESP (grant 2013/07375-0).

CHAPTER

8

CONCLUSION AND FINAL REMARKS

In this thesis, we sought to answer the three main research questions presented in section 1.2. To that end, we formulated hypotheses that guided us in the publications that compose each chapter that comprehends the present study. Most of these research questions concern reducing the computational costs of the incremental decision trees, decision rules, and ensemble models, primarily when considering regression tasks.

In Chapter 2, we evaluated different strategies to create incremental ensemble regressors and how such methods impact predictive and computational performance. We also introduced SRP-Reg, which combines instance re-sampling and feature sub-sampling to create diverse and accurate regressors. We show that, in regression tasks, passive concept drift detection techniques can be as effective as the active ones, e.g., resetting the base learners at predefined intervals rather than relying upon concept drift detectors. We also verified that combining the prediction of model trees using the median value leads to improved predictive performance. Notwithstanding, the costs of the base models in regression tasks are a limiting factor to the application of ensembles.

Chapter 3 attempted to address this issue by employing the correlation between features and the target as a heuristic to guide splits. Our proposal is dubbed 2CS, and it can speed up Hoeffding Tree regressor (HTR) construction at the cost of decreasing predictive performance at some times. Memory usage is also slightly increased. Due to these limitations, we realized that we should focus on decreasing their construction costs directly instead of adding additional heuristics to the HTRs.

Hence, in Chapter 4, we focused on creating an improved mechanism to perform splits attempts in HTRs. Besides that, we also introduce improved formulae to keep incremental variance estimators, which the HTRs use. Our quantization-based split strategy and the improved variance estimators significantly decreased the time needed to calculate the split decisions while also using significantly fewer memory resources. The split points found by our proposal, QO, are on par with the exhaustive strategy commonly applied in the literature. In Chapter 5, we

expand the work of the previous chapter and apply QO in HTRs to assess the effectiveness of our proposal in the situations it was designed to work. We also evaluate the possibility of creating multi-branch numerical HTRs that are as efficient and effective as the strictly binary ones while being shallow and broader than the traditional HTRs. We show that the QO-powered HTRs are significantly faster than the original ones while using a significantly reduced memory footprint and still delivering comparable predictive performance.

Although the creation of individual HTRs, and decision rule regressors, by consequence, was sped up significantly, joining the regressors into ensembles was still problematic. The joint computational performance of multiple HTRs constructed deterministically might not be fast enough for real-world applications. Therefore, in Chapter 6, we proposed OXT and incremental ensemble whose HTRs are built randomly. Besides, OXT explores sub-bagging to induce diversity in the ensemble members. Our proposal is significantly faster than the state-of-the-art ensembles of HTRs, uses substantially fewer memory resources, and is generally more accurate than the competitors.

Thus, by combining the insights obtained in Chapter 2 and Chapter 3, and the gains obtained in individual HTR building (Chapter 4 and Chapter 5) and the creation of incremental ensemble regressors (Chapter 6), we believe we addressed the research questions that concerned decision tree, decision rules, and tree ensemble building.

Lastly, we tackled another primary concern in Chapter 7, which is incremental nearest neighbor search. In this chapter, we propose SWINN, a graph-based incremental search index that can significantly speed up nearest neighbor search in sliding window buffers. SWINN works with arbitrary distance measures and delivers competitive search recall results. We perform realistic case studies to assess the performance of SWINN against popular OML models. SWINN can process significantly more instances than the traditional nearest neighbor algorithms while delivering comparative predictive performance.

8.1 Limitations and future work

This thesis was primarily focused on the theoretical development of solutions for speeding up and reducing the memory footprint of OML models. The effectiveness of the developed solutions was assessed via empirical evaluation setups and asymptotic algorithm analysis. In all of our proposals, extending the number of compared datasets would be beneficial to better comprehend the impact of our proposal on real-world applications. Real-world applications where the algorithms would possibly process data indefinitely are crucial scenarios for evaluation that are not considered in this thesis aside from a theoretical perspective. Besides that, we did not investigate the impact of label delay (GOMES *et al.*, 2022) on our proposals. In realistic supervised OML scenarios, labels are not expected to be available shortly after predicting a new instance in most applications, since the labeling process might be expensive and too slow.

Instead, the labels have an expected delay in arriving. The learning capabilities of the OML are clearly impacted by label scarcity.

As for future research directions, each chapter delineates open challenges concerning each associated proposal. However, when looking at the thesis as a collective research, we can envision possible future works that would join the published results. A first and direct step in future work would be joining some of the proposed strategies into a single algorithm. For instance, we can build ensembles using the findings from Chapter 2 and powered by the trees defined in Chapter 4 and Chapter 5. The heuristic defined in Chapter 3 could also be applied to the combined ensemble. We would compare this resulting ensemble algorithm against the one presented in Chapter 6.

Concerning the contents of Chapter 7, we intend to explore other tasks that could be benefited from the speed-ups brought by our proposal. Examples include anomaly detection, structured multi-output tasks (XU *et al.*, 2019), and clustering. This last task can be performed by setting a small number of neighbors to construct the search graph. Thus, the resulting search index ought to have multiple sub-graphs, i.e., multiple clusters of nodes.

BIBLIOGRAPHY

ALMEIDA, E.; FERREIRA, C.; GAMA, J. Adaptive model rules from data streams. In: SPRINGER. **Joint European Conference on Machine Learning and Knowledge Discovery in Databases**. [S.l.], 2013. p. 480–492. Citations on pages 42 and 72.

ANDONI, A.; INDYK, P.; LAARHOVEN, T.; RAZENSHTEYN, I.; SCHMIDT, L. Practical and optimal lsh for angular distance. **arXiv preprint arXiv:1509.02897**, 2015. Citations on pages 126 and 128.

AUMÜLLER, M.; BERNHARDSSON, E.; FAITHFULL, A. Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. **Information Systems**, Elsevier, v. 87, p. 101374, 2020. Citations on pages 126, 127, and 129.

BAHRI, M.; BIFET, A.; GAMA, J.; GOMES, H. M.; MANIU, S. Data stream analysis: Foundations, major tasks and tools. **Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery**, Wiley Online Library, v. 11, n. 3, p. e1405, 2021. Citation on page 104.

BARDDAL, J. P.; ENEMBRECK, F. Learning regularized hoeffding trees from data streams. In: ACM. **Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing**. [S.l.], 2019. p. 574–581. Citations on pages 57 and 60.

BARROS, R. S. M. de; SANTOS, S. G. T. d. C.; BARDDAL, J. P. Evaluating k-nn in the classification of data streams with concept drift. **arXiv preprint arXiv:2210.03119**, 2022. Citation on page 126.

BIFET, A. Classifier concept drift detection and the illusion of progress. In: SPRINGER. **ICAISC**. [S.l.], 2017. p. 715–725. Citation on page 46.

BIFET, A.; GAVALDA, R. Learning from time-changing data with adaptive windowing. In: SIAM. **Proceedings of the 2007 SIAM international conference on data mining**. [S.l.], 2007. p. 443–448. Citations on pages 42, 44, and 114.

BIFET, A.; GAVALDÀ, R. Adaptive learning from evolving data streams. In: SPRINGER. **International Symposium on Intelligent Data Analysis**. [S.l.], 2009. p. 249–260. Citations on pages 29, 44, 58, 91, 92, and 150.

BIFET, A.; GAVALDÀ, R.; HOLMES, G.; PFAHRINGER, B. **Machine Learning for Data Streams with Practical Examples in MOA**. [S.l.]: MIT Press, 2018. <https://moa.cms.waikato.ac.nz/book/>. Citations on pages 29, 48, 72, 86, and 88.

BIFET, A.; HOLMES, G.; PFAHRINGER, B. Leveraging bagging for evolving data streams. In: SPRINGER. **Joint European conference on machine learning and knowledge discovery in databases**. [S.l.], 2010. p. 135–150. Citations on pages 43, 105, 106, 107, 115, and 123.

BISHOP, C. M. **Pattern recognition and machine learning**. [S.l.]: springer, 2006. Citations on pages 31 and 60.

BLUM, A.; KALAI, A.; LANGFORD, J. Beating the hold-out: Bounds for k-fold and progressive cross-validation. In: **Proceedings of the twelfth annual conference on Computational learning theory**. [S.l.: s.n.], 1999. p. 203–208. Citation on page 141.

BRATIĆ, B.; HOULE, M. E.; KURBALIJA, V.; ORIA, V.; RADOVANOVIĆ, M. Nn-descent on high-dimensional data. In: **Proceedings of the 8th International Conference on Web Intelligence, Mining and Semantics**. [S.l.: s.n.], 2018. p. 1–8. Citations on pages 131, 142, 143, 147, and 153.

____. The influence of hubness on nn-descent. **International Journal on Artificial Intelligence Tools**, World Scientific, v. 28, n. 06, p. 1960002, 2019. Citations on pages 131, 142, 143, 147, and 153.

BRAZDIL, P.; CARRIER, C. G.; SOARES, C.; VILALTA, R. **Metalearning: Applications to data mining**. [S.l.]: Springer Science & Business Media, 2008. Citation on page 66.

BREIMAN, L. Bagging predictors. **Machine learning**, Springer, v. 24, n. 2, p. 123–140, 1996. Citations on pages 32, 40, 43, 104, and 106.

____. Random forests. **Machine Learning**, v. 45, n. 1, p. 5–32, 2001. Citations on pages 31, 32, 40, and 43.

BREIMAN, L.; FRIEDMAN, J. H.; OLSHEN, R. A.; STONE, C. J. **Classification and regression trees**. [S.l.]: Routledge, 2017. Citations on pages 31, 32, 59, and 104.

BÜHLMANN, P. L. Bagging, subagging and bragging for improving some prediction algorithms. In: **Research Seminar für Statistik, Eidgenössische Technische Hochschule (ETH)**. [S.l.: s.n.], 2003. v. 113. Citation on page 44.

CAI, Y.; XU, W.; ZHANG, F. ikd-tree: An incremental kd tree for robotic applications. **arXiv preprint arXiv:2102.10808**, 2021. Citations on pages 127, 128, and 131.

CANO, A.; KRAWCZYK, B. Rose: robust online self-adjusting ensemble for continual learning on imbalanced drifting data streams. **Machine Learning**, Springer, p. 1–39, 2022. Citation on page 30.

CHAN, T. F.; GOLUB, G. H.; LEVEQUE, R. J. Updating formulae and a pairwise algorithm for computing sample variances. In: SPRINGER. **COMPSTAT 1982 5th Symposium held at Toulouse 1982**. [S.l.], 1982. p. 30–41. Citations on pages 74, 76, and 77.

CHEN, T.; GUESTRIN, C. Xgboost: A scalable tree boosting system. In: ACM. **Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining**. [S.l.], 2016. p. 785–794. Citation on page 86.

CRAMMER, K.; DEKEL, O.; KESHET, J.; SHALEV-SHWARTZ, S.; SINGER, Y. Online passive-aggressive algorithms. **Journal of Machine Learning Research**, v. 7, n. 19, p. 551–585, 2006. Available: <http://jmlr.org/papers/v7/crammer06a.html>. Citations on pages 92 and 114.

DATAR, M.; IMMORLICA, N.; INDYK, P.; MIRROKNI, V. S. Locality-sensitive hashing scheme based on p-stable distributions. In: **Proceedings of the twentieth annual symposium on Computational geometry**. [S.l.: s.n.], 2004. p. 253–262. Citations on pages 77, 126, and 128.

DEMŠAR, J. Statistical comparisons of classifiers over multiple data sets. **Journal of Machine learning research**, v. 7, n. Jan, p. 1–30, 2006. Citations on pages 63, 81, and 92.

DOMINGOS, P. Every model learned by gradient descent is approximately a kernel machine. **arXiv preprint arXiv:2012.00152**, 2020. Citation on page 126.

DOMINGOS, P.; HULTEN, G. Mining high-speed data streams. In: **Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining**. Boston, MA, USA: ACM, 2000. p. 71–80. Citations on pages 30, 32, 41, 46, 56, 72, 73, 86, and 104.

DONG, W.; MOSES, C.; LI, K. Efficient k-nearest neighbor graph construction for generic similarity measures. In: **Proceedings of the 20th international conference on World wide web**. [S.l.: s.n.], 2011. p. 577–586. Citations on pages 127, 129, 130, 131, 142, and 153.

DONG, X.; YU, Z.; CAO, W.; SHI, Y.; MA, Q. A survey on ensemble learning. **Frontiers of Computer Science**, Springer, v. 14, n. 2, p. 241–258, 2020. Citation on page 106.

DUARTE, J.; GAMA, J. Multi-target regression from high-speed data streams with adaptive model rules. In: **Data Science and Advanced Analytics (DSAA), 2015. 36678 2015. IEEE International Conference on**. Campus des Cordeliers, Paris, France: IEEE, 2015. p. 1–10. Citations on pages 32 and 62.

DUARTE, J.; GAMA, J.; BIFET, A. Adaptive model rules from high-speed data streams. **ACM Transactions on Knowledge Discovery from Data (TKDD)**, ACM New York, NY, USA, v. 10, n. 3, p. 1–22, 2016. Citations on pages 32, 87, 88, 92, and 114.

FAN, W.; BIFET, A. Mining big data: current status, and forecast to the future. **ACM sIGKDD Explorations Newsletter**, ACM, v. 14, n. 2, p. 1–5, 2013. Citation on page 56.

FERNÁNDEZ-DELGADO, M.; CERNADAS, E.; BARRO, S.; AMORIM, D. Do we need hundreds of classifiers to solve real world classification problems? **The journal of machine learning research**, JMLR. org, v. 15, n. 1, p. 3133–3181, 2014. Citation on page 86.

FINCH, T. Incremental calculation of weighted mean and variance. **University of Cambridge**, v. 4, n. 11-5, p. 41–42, 2009. Citations on pages 74, 75, and 76.

FRIEDMAN, J. H. Greedy function approximation: a gradient boosting machine. **Annals of statistics**, JSTOR, p. 1189–1232, 2001. Citation on page 106.

FRIEDMAN, J. H.; BENTLEY, J. L.; FINKEL, R. A. An algorithm for finding best matches in logarithmic expected time. **ACM Transactions on Mathematical Software (TOMS)**, ACM New York, NY, USA, v. 3, n. 3, p. 209–226, 1977. Citation on page 128.

GABRIEL, K. R. The biplot graphic display of matrices with application to principal component analysis. **Biometrika**, Oxford University Press, v. 58, n. 3, p. 453–467, 1971. Citations on pages 68 and 93.

GAMA, J. **Knowledge discovery from data streams**. [S.l.]: Chapman and Hall/CRC, 2010. Citations on pages 29, 31, 32, 56, 57, 58, 59, 60, 63, 72, 86, and 91.

GAMA, J.; SEBASTIAO, R.; RODRIGUES, P. P. Issues in evaluation of stream learning algorithms. In: **Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining**. [S.l.: s.n.], 2009. p. 329–338. Citation on page 141.

GAMA, J.; ŽLIOBAITĖ, I.; BIFET, A.; PECHENIZKIY, M.; BOUCHACHIA, A. A survey on concept drift adaptation. **ACM computing surveys (CSUR)**, ACM New York, NY, USA, v. 46, n. 4, p. 1–37, 2014. Citation on page 29.

GARG, S.; SINHA, S.; KAR, A. K.; MANI, M. A review of machine learning applications in human resource management. **International Journal of Productivity and Performance Management**, Emerald Publishing Limited, 2021. Citation on page 29.

GEURTS, P.; ERNST, D.; WEHENKEL, L. Extremely randomized trees. **Machine learning**, Springer, v. 63, n. 1, p. 3–42, 2006. Citations on pages 32, 105, and 114.

GOMES, H. M.; BARDDAL, J. P.; ENEMBRECK, F.; BIFET, A. A survey on ensemble learning for data stream classification. **ACM Computing Surveys (CSUR)**, ACM, v. 50, n. 2, p. 23, 2017. Citations on pages 32, 40, 43, 56, 104, and 106.

GOMES, H. M.; BARDDAL, J. P.; FERREIRA, L. E. B.; BIFET, A. Adaptive random forests for data stream regression. In: **26th European Symposium on Artificial Neural Networks, ESANN 2018, Bruges, Belgium, April 25-27, 2018**. [s.n.], 2018. Available: <http://www.elen.ucl.ac.be/Proceedings/esann/esannpdf/es2018-183.pdf>. Citations on pages 40, 41, 42, 43, 44, 45, 56, 58, 86, 87, 88, 104, 105, 107, 112, 113, 115, and 150.

GOMES, H. M.; BIFET, A.; READ, J.; BARDDAL, J. P.; ENEMBRECK, F.; PFHARINGER, B.; HOLMES, G.; ABDESSALEM, T. Adaptive random forests for evolving data stream classification. **Machine Learning**, Springer, v. 106, n. 9-10, p. 1469–1495, 2017. Citations on pages 40, 41, 42, 43, 44, 45, 105, 106, 107, 108, 113, and 150.

GOMES, H. M.; GRZENDA, M.; MELLO, R.; READ, J.; NGUYEN, M. H. L.; BIFET, A. A survey on semi-supervised learning for delayed partially labelled data streams. **ACM Computing Surveys**, ACM New York, NY, v. 55, n. 4, p. 1–42, 2022. Citations on pages 113, 124, and 156.

GOMES, H. M.; MONTIEL, J.; MASTELINI, S. M.; PFAHRINGER, B.; BIFET, A. On ensemble techniques for data stream regression. In: IEEE. **2020 International Joint Conference on Neural Networks (IJCNN)**. [S.l.], 2020. p. 1–8. Citations on pages 35, 86, 88, 104, 105, 106, 108, 109, 112, 114, 117, and 118.

GOMES, H. M.; READ, J.; BIFET, A. Streaming random patches for evolving data stream classification. In: IEEE. **ICDM**. [S.l.], 2019. Citations on pages 42, 43, 45, and 46.

GOMES, H. M.; READ, J.; BIFET, A.; DURRANT, R. J. Learning from evolving data streams through ensembles of random patches. **Knowledge and Information Systems**, Springer, p. 1–29, 2021. Citations on pages 30, 37, 104, 105, 106, and 108.

GOUK, H.; PFAHRINGER, B.; FRANK, E. Stochastic gradient trees. **arXiv preprint arXiv:1901.07777**, 2019. Citations on pages 72, 73, and 88.

GRZENDA, M.; GOMES, H. M.; BIFET, A. Delayed labelling evaluation for data streams. **Data Mining and Knowledge Discovery**, Springer, p. 1–30, 2019. Citation on page 57.

HO, T. K. Random decision forests. In: IEEE. **International Conference on Document Analysis and Recognition**. [S.l.], 1995. v. 1. Citations on pages 31, 40, 42, and 43.

____. Nearest neighbors in random subspaces. In: SPRINGER. **IAPR International Workshops on Statistical Techniques in Pattern Recognition and Structural and Syntactic Pattern Recognition**. [S.l.], 1998. Citation on page 46.

HOEFFDING, W. Probability inequalities for sums of bounded random variables. **Journal of the American statistical association**, Taylor & Francis Group, 1963. Citations on pages 32, 41, and 72.

HOENS, T. R.; CHAWLA, N. V.; POLIKAR, R. Heuristic updatable weighted random subspaces for non-stationary environments. In: IEEE. **ICDM**. [S.l.], 2011. Citations on pages 42 and 43.

HOTHORN, T.; HORNIK, K.; ZEILEIS, A. Unbiased recursive partitioning: A conditional inference framework. **Journal of Computational and Graphical statistics**, Taylor & Francis, v. 15, n. 3, p. 651–674, 2006. Citation on page 59.

HULTEN, G.; SPENCER, L.; DOMINGOS, P. Mining time-changing data streams. In: **Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining**. [S.l.]: ACM, 2001. p. 97–106. Citations on pages 57 and 58.

IKONOMOVSKA, E.; GAMA, J.; DŽEROSKI, S. Incremental multi-target model trees for data streams. In: ACM. **Proceedings of the 2011 ACM symposium on applied computing**. [S.l.], 2011. p. 988–993. Citations on pages 32, 56, and 58.

____. Learning model trees from evolving data streams. **Data mining and knowledge discovery**, Springer, v. 23, n. 1, p. 128–168, 2011. Citations on pages 30, 32, 41, 44, 46, 56, 58, 59, 60, 61, 62, 72, 73, 74, 86, 87, 88, 92, 104, 105, 110, 112, 114, and 150.

____. Online tree-based ensembles and option trees for regression on evolving data streams. **Neurocomputing**, Elsevier, v. 150, p. 458–470, 2015. Citations on pages 32, 42, 43, 46, 56, 57, 58, 59, 86, and 88.

IKONOMOVSKA, E.; GAMA, J.; ZENKO, B.; DZEROSKI, S. Speeding-up hoeffding-based regression trees with options. In: CITESEER. **ICML**. [S.l.], 2011. p. 537–544. Citation on page 41.

JEGOU, H.; DOUZE, M.; SCHMID, C. Product quantization for nearest neighbor search. **IEEE transactions on pattern analysis and machine intelligence**, IEEE, v. 33, n. 1, p. 117–128, 2010. Citations on pages 126, 129, and 131.

JO, J.; SEO, J.; FEKETE, J.-D. Panene: A progressive algorithm for indexing and querying approximate k-nearest neighbors. **IEEE transactions on visualization and computer graphics**, IEEE, v. 26, n. 2, p. 1347–1360, 2018. Citations on pages 127, 128, and 131.

KE, G.; MENG, Q.; FINLEY, T.; WANG, T.; CHEN, W.; MA, W.; YE, Q.; LIU, T.-Y. Lightgbm: A highly efficient gradient boosting decision tree. In: **Advances in neural information processing systems**. [S.l.: s.n.], 2017. p. 3146–3154. Citation on page 86.

KNUTH, D. E. **Art of computer programming, volume 2: Seminumerical algorithms**. [S.l.]: Addison-Wesley Professional, 2014. Citations on pages 74, 75, and 76.

KOHAVI, R.; KUNZ, C. Option decision trees with majority votes. In: **ICML**. [S.l.: s.n.], 1997. v. 97, p. 161–169. Citations on pages 31 and 32.

KORYCKI, Ł.; KRAWCZYK, B. Adaptive deep forest for online learning from drifting data streams. **arXiv preprint arXiv:2010.07340**, 2020. Citations on pages 30 and 107.

KRAWCZYK, B.; MINKU, L. L.; GAMA, J. a.; STEFANOWSKI, J.; WOŹNIAK, M. Ensemble learning for data stream analysis: A survey. **Information Fusion**, Elsevier, v. 37, p. 132–156, 2017. Citations on pages 30, 32, 56, 63, 72, 104, and 106.

KRUSKAL, J. B. On the shortest spanning subtree of a graph and the traveling salesman problem. **Proceedings of the American Mathematical society**, JSTOR, v. 7, n. 1, p. 48–50, 1956. Citation on page 138.

LIBBRECHT, M. W.; NOBLE, W. S. Machine learning applications in genetics and genomics. **Nature Reviews Genetics**, Nature Publishing Group, v. 16, n. 6, p. 321–332, 2015. Citation on page 29.

LIU, Q.; ZHANG, J.; LIAN, D.; GE, Y.; MA, J.; CHEN, E. Online additive quantization. In: **Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining**. [S.l.: s.n.], 2021. p. 1098–1108. Citations on pages 127 and 129.

LOH, W.-Y. Classification and regression trees. **Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery**, Wiley Online Library, v. 1, n. 1, p. 14–23, 2011. Citations on pages 31 and 32.

LOSING, V.; HAMMER, B.; WERSING, H. Tackling heterogeneous concept drift with the self-adjusting memory (sam). **Knowledge and Information Systems**, Springer, v. 54, n. 1, p. 171–201, 2018. Citation on page 127.

LOSING, V.; HAMMER, B.; WERSING, H.; BIFET, A. Randomizing the self-adjusting memory for enhanced handling of concept drift. In: IEEE. **2020 International Joint Conference on Neural Networks (IJCNN)**. [S.l.], 2020. p. 1–8. Citation on page 127.

LOUPPE, G.; GEURTS, P. Ensembles on random patches. In: SPRINGER. **ECML**. [S.l.], 2012. p. 346–361. Citations on pages 42 and 43.

LU, J.; LIU, A.; DONG, F.; GU, F.; GAMA, J.; ZHANG, G. Learning under concept drift: A review. **IEEE Transactions on Knowledge and Data Engineering**, IEEE, v. 31, n. 12, p. 2346–2363, 2018. Citations on pages 29 and 104.

LUONG, A. V.; NGUYEN, T. T.; LIEW, A. W.-C. Streaming active deep forest for evolving data stream classification. **arXiv preprint arXiv:2002.11816**, 2020. Citation on page 107.

MALKOV, Y. A.; YASHUNIN, D. A. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. **IEEE transactions on pattern analysis and machine intelligence**, IEEE, v. 42, n. 4, p. 824–836, 2018. Citation on page 129.

MANAPRAGADA, C.; WEBB, G. I.; SALEHI, M. Extremely fast decision tree. In: **Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining**. New York, NY, USA: ACM, 2018. (KDD '18), p. 1953–1962. ISBN 978-1-4503-5552-0. Available: <http://doi.acm.org/10.1145/3219819.3220005>. Citation on page 91.

MASTELINI, S. M.; CARVALHO, A. C. P. d. L. F. de. Using dynamical quantization to perform split attempts in online tree regressors. **Pattern Recognition Letters**, Elsevier, v. 145, p. 37–42, 2021. Citations on pages 34, 36, 86, 87, 88, 89, 105, 110, 112, 114, and 119.

MASTELINI, S. M.; CARVALHO, A. C. Ponce de Leon Ferreira de. 2cs: correlation-guided split candidate selection in hoeffding tree regressors. In: SPRINGER. **Brazilian Conference on Intelligent Systems**. [S.l.], 2020. p. 337–351. Citations on pages 34 and 35.

MASTELINI, S. M.; CASSAR, D. R.; ALCOBAÇA, E.; BOTARI, T.; CARVALHO, A. C. de; ZANOTTO, E. D. Machine learning unveils composition-property relationships in chalcogenide glasses. **Acta Materialia**, Elsevier, v. 240, p. 118302, 2022. Citation on page 29.

MASTELINI, S. M.; JR, S. B.; CARVALHO, A. C. P. d. de; FERREIRA, L. Online multi-target regression trees with stacked leaf models. **arXiv preprint arXiv:1903.12483**, 2019. Citations on pages 56, 58, and 59.

MASTELINI, S. M.; MONTIEL, J.; GOMES, H. M.; BIFET, A.; PFAHRINGER, B.; CARVALHO, A. C. de. Fast and lightweight binary and multi-branch Hoeffding Tree Regressors. In: IEEE. **2021 International Conference on Data Mining Workshops (ICDMW)**. [S.l.], 2021. p. 380–388. Citations on pages 34 and 36.

MASTELINI, S. M.; NAKANO, F. K.; VENS, C.; FERREIRA, A. C. P. de L. *et al.* Online extra trees regressor. **IEEE Transactions on Neural Networks and Learning Systems**, IEEE, 2022. Citations on pages 34 and 37.

MATSUI, Y.; UCHIDA, Y.; JéGOU, H.; SATOH, S. A survey of product quantization. **ITE Transactions on Media Technology and Applications**, v. 6, n. 1, p. 2–10, 2018. Citations on pages 126, 129, and 131.

MENDES-MOREIRA, J.; SOARES, C.; JORGE, A. M.; SOUSA, J. F. D. Ensemble approaches for regression: A survey. **Acm computing surveys (csur)**, ACM, v. 45, n. 1, p. 10, 2012. Citations on pages 42 and 43.

MONTIEL, J.; BIFET, A.; LOSING, V.; READ, J.; ABDESSALEM, T. Learning fast and slow: A unified batch/stream framework. In: **2018 IEEE International Conference on Big Data (Big Data)**. [S.l.: s.n.], 2018. p. 1065–1072. Citation on page 45.

MONTIEL, J.; HALFORD, M.; MASTELINI, S. M.; BOLMIER, G.; SOURTY, R.; VAYSSE, R.; ZOUITINE, A.; GOMES, H. M.; READ, J.; ABDESSALEM, T.; BIFET, A. River: machine learning for streaming data in python. **Journal of Machine Learning Research**, v. 22, n. 110, p. 1–8, 2021. Available: <http://jmlr.org/papers/v22/20-1380.html>. Citations on pages 34, 38, 90, 91, 112, 114, 140, and 149.

MONTIEL, J.; READ, J.; BIFET, A.; ABDESSALEM, T. Scikit-multiflow: a multi-output streaming framework. **The Journal of Machine Learning Research**, JMLR. org, v. 19, n. 1, p. 2915–2914, 2018. Citations on pages 48, 58, and 61.

MOURTADA, J.; GAÏFFAS, S.; SCORNET, E. AMF: Aggregated Mondrian forests for online learning. **Journal of the Royal Statistical Society: Series B (Statistical Methodology)**, Wiley Online Library, v. 83, n. 3, p. 505–533, 2021. Citation on page 107.

MOUSS, H.; MOUSS, D.; MOUSS, N.; SEFOUHI, L. Test of page-hinckley, an approach for fault detection in an agro-alimentary production system. In: **Asian Control Conference**. [S.l.: s.n.], 2004. v. 2. Citation on page 42.

OMOHUNDRO, S. M. **Five balltree construction algorithms**. [S.l.]: International Computer Science Institute Berkeley, 1989. Citation on page 128.

OSOJNIK, A.; PANOV, P.; DŽEROSKI, S. Multi-label classification via multi-target regression on data streams. **Machine Learning**, Springer, v. 106, n. 6, p. 745–770, 2017. Citations on pages 32 and 88.

____. Tree-based methods for online multi-target regression. **Journal of Intelligent Information Systems**, Springer, v. 50, n. 2, p. 315–339, 2018. Citations on pages 56, 57, 58, 59, 61, 62, 63, 72, 73, 88, and 114.

____. Incremental predictive clustering trees for online semi-supervised multi-target regression. **Machine Learning**, Springer, v. 109, n. 11, p. 2121–2139, 2020. Citation on page 88.

OZA, N. C.; RUSSELL, S. Experimental comparisons of online and batch versions of bagging and boosting. In: **ACM SIGKDD**. [S.l.: s.n.], 2001. Citations on pages 40 and 43.

OZA, N. C.; RUSSELL, S. J. Online bagging and boosting. In: PMLR. **International Workshop on Artificial Intelligence and Statistics**. [S.l.], 2001. p. 229–236. Citations on pages 105, 106, 107, 108, and 123.

PFAHRINGER, B.; HOLMES, G.; KIRKBY, R. New options for hoeffding trees. In: SPRINGER. **Australasian Joint Conference on Artificial Intelligence**. [S.l.], 2007. p. 90–99. Citations on pages 31, 32, and 72.

____. Handling numeric attributes in hoeffding trees. In: SPRINGER. **Pacific-Asia Conference on Knowledge Discovery and Data Mining**. [S.l.], 2008. p. 296–307. Citations on pages 30, 73, and 152.

RUSSELL, S. J.; NORVIG, P. **Artificial intelligence: a modern approach**. [S.l.]: Malaysia; Pearson Education Limited,, 2016. Citations on pages 29 and 31.

RUTKOWSKI, L.; PIETRUCZUK, L.; DUDA, P.; JAWORSKI, M. Decision trees for mining data streams based on the mcdiarmid's bound. **IEEE Transactions on Knowledge and Data Engineering**, IEEE, v. 25, n. 6, p. 1272–1279, 2012. Citation on page 72.

SAGI, O.; ROKACH, L. Ensemble learning: A survey. **Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery**, Wiley Online Library, v. 8, n. 4, p. e1249, 2018. Citations on pages 30, 104, and 106.

SALEHI-MOGHADDAMI, N.; YAZDI, H. S.; POOSTCHI, H. Correlation based splitting criterion in multi branch decision tree. **Central European Journal of Computer Science**, Springer, v. 1, n. 2, p. 205–220, 2011. Citation on page 59.

SARKER, I. H. Machine learning: Algorithms, real-world applications and research directions. **SN Computer Science**, Springer, v. 2, n. 3, p. 1–21, 2021. Citation on page 29.

SHIMOMURA, L. C.; OYAMADA, R. S.; VIEIRA, M. R.; KASTER, D. S. A survey on graph-based methods for similarity searches in metric spaces. **Information Systems**, Elsevier, v. 95, p. 101507, 2021. Citations on pages 126, 127, and 129.

SOUSA, R.; GAMA, J. Multi-label classification from high-speed data streams with adaptive model rules and random rules. **Progress in Artificial Intelligence**, Springer, v. 7, n. 3, p. 177–187, 2018. Citation on page 88.

TORGO, L. Functional models for regression tree leaves. In: CITESEER. **ICML**. [S.l.], 1997. v. 97, p. 385–393. Citation on page 32.

WANG, A.; WAN, G.; CHENG, Z.; LI, S. An incremental extremely random forest classifier for online learning and tracking. In: IEEE. **2009 16th ieee international conference on image processing (icip)**. [S.l.], 2009. p. 1449–1452. Citation on page 107.

WANG, W.; SIAU, K. Artificial intelligence, machine learning, automation, robotics, future of work and future of humanity: A review and research agenda. **Journal of Database Management (JDM)**, IGI Global, v. 30, n. 1, p. 61–79, 2019. Citation on page 29.

WEST, D. B. *et al.* **Introduction to graph theory**. [S.l.]: Prentice hall Upper Saddle River, NJ, 1996. Citation on page 31.

XU, D.; SHI, Y.; TSANG, I. W.; ONG, Y.-S.; GONG, C.; SHEN, X. Survey on multi-output learning. **IEEE transactions on neural networks and learning systems**, IEEE, v. 31, n. 7, p. 2409–2429, 2019. Citation on page 157.

XU, D.; TSANG, I. W.; ZHANG, Y. Online product quantization. **IEEE Transactions on Knowledge and Data Engineering**, IEEE, v. 30, n. 11, p. 2185–2198, 2018. Citations on pages 127 and 129.

YATES, D.; ISLAM, M. Z. Fastforest: Increasing random forest processing speed while maintaining accuracy. **Information Sciences**, Elsevier, v. 557, p. 130–152, 2021. Citations on pages 107 and 108.

ZAMAN, F.; HIROSE, H. Effect of subsampling rate on subbagging and related ensembles of stable classifiers. In: SPRINGER. **International Conference on Pattern Recognition and Machine Intelligence**. [S.l.], 2009. p. 44–49. Citations on pages 107, 108, and 115.

ZHOU, Z.-H.; YU, Y. Adapt bagging to nearest neighbor classifiers. **Journal of Computer Science and Technology**, Springer, v. 20, n. 1, p. 48–54, 2005. Citation on page 46.

APPENDIX

A

SUPPLEMENTARY MATERIAL FOR: "SWINN: EFFICIENT NEAREST NEIGHBOR SEARCH IN SLIDING WINDOWS USING GRAPHS"

A.1 Impact of K in the search graph for windows of 5000 instances

We present the results obtained when varying the values of K and using $L = 5000$ in Figure 34.

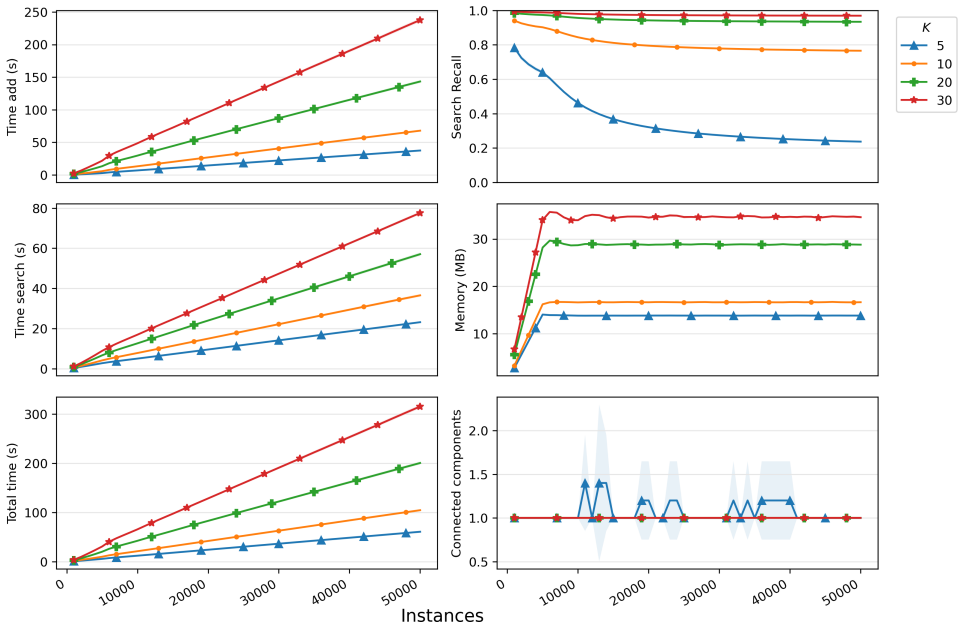


Figure 34 – Results obtained when varying K and using sliding windows of length 5000.

A.2 Impact of $\texttt{max}_c$ in the search graph for windows of 1000 instances

Figure 35 presents the results obtained when varying the values of $\texttt{max}_c$ and using $L = 1000$.

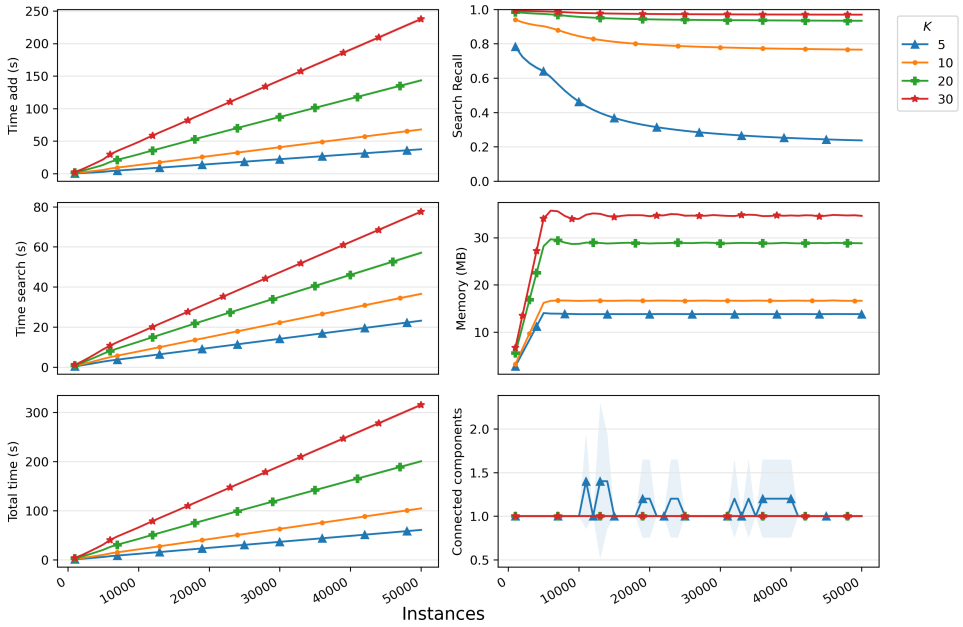


Figure 35 – Results obtained when varying $\texttt{max}_c$ and using sliding windows of length 1000.

A.3 The impact of $\texttt{prune}_{\textbf{prob}}$ in the search graph for windows of 1000instances

Figure 36 presents the results obtained when varying the values of $\texttt{prune}_{\text{prob}}$ and using $L = 1000$.

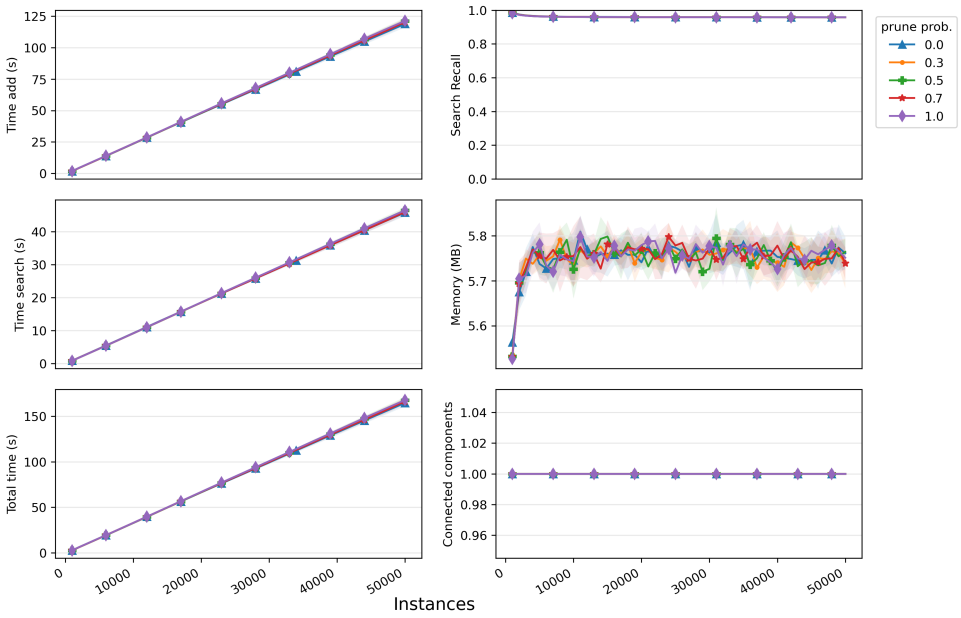


Figure 36 – Results obtained when varying $\texttt{prune}_{\text{prob}}$ and using sliding windows of length 1000.

APPENDIX

B

LIST OF ADDITIONAL PUBLICATIONS MADE DURING THE THESIS DEVELOPMENT

- MASTELINI, Saulo Martiello et al. Benchmarking multi-target regression methods. In: **2018 7th Brazilian Conference on Intelligent Systems (BRACIS).** IEEE, 2018. p. 396-401.

- DA COSTA, Victor G. Turrisi et al. Making data stream classification tree-based ensembles lighter. In: **2018 7th Brazilian Conference on Intelligent Systems (BRACIS).** IEEE, 2018. p. 480-485.

- GERONIMO, Bruna Caroline et al. Computer vision system and near-infrared spectroscopy for identification and classification of chicken with wooden breast, and physicochemical and technological characterization. **Infrared Physics & Technology**, v. 96, p. 303-310, 2019.

- MASTELINI, Saulo Martiello et al. Multi-output tree chaining: An interpretative modelling and lightweight multi-target approach. **Journal of Signal Processing Systems**, v. 91, n. 2, p. 191-215, 2019.

- SANTANA, Everton Jose et al. Stock portfolio prediction by multi-target decision support. **iSys-Brazilian Journal of Information Systems**, v. 12, n. 1, p. 05-27, 2019.

- KATO, Talita et al. White striping degree assessment using computer vision system and consumer acceptance test. **Asian-Australasian Journal of Animal Sciences**, v. 32, n. 7, p. 1015, 2019.

- DA COSTA, Victor G. Turrisi et al. Online local boosting: Improving performance in online decision trees. In: **2019 8th Brazilian Conference on Intelligent Systems (BRACIS).** IEEE, 2019. p. 132-137.

- AGUIAR, Gabriel J. et al. Towards meta-learning for multi-target regression problems. In: **2019 8th Brazilian Conference on Intelligent Systems (BRACIS).** IEEE, 2019. p. 377-382.

- CAMPOS, Gabriel Fillipe Centini et al. Machine learning hyperparameter selection for contrast limited adaptive histogram equalization. **EURASIP Journal on Image and Video Processing**, v. 2019, n. 1, p. 1-18, 2019.

- AGUIAR, Gabriel Jonas et al. A meta-learning approach for selecting image segmentation algorithm. **Pattern Recognition Letters**, v. 128, p. 480-487, 2019.

- ALCOBACA, Edesio et al. Explainable machine learning algorithms for predicting glass transition temperatures. **Acta Materialia**, v. 188, p. 92-100, 2020.

- MASTELINI, Saulo Martiello et al. DSTARS: a multi-target deep structure for tracking asynchronous regressor stacking. **Applied Soft Computing**, v. 91, p. 106215, 2020.

- JUNIOR, Sylvio Barbon et al. Multi-target prediction of wheat flour quality parameters with near infrared spectroscopy. **Information processing in agriculture**, v. 7, n. 2, p. 342-354, 2020.

- MONTIEL, Jacob et al. River: machine learning for streaming data in Python. **Journal of Machine Learning Research**, v. 22, n. 110, p. 1-8, 2021.

- SANTANA, Everton Jose et al. Improved prediction of soil properties with multi-target stacked generalisation on EDXRF spectra. **Chemometrics and Intelligent Laboratory Systems**, v. 209, p. 104231, 2021.

- CASSAR, Daniel R. et al. Predicting and interpreting oxide glass properties by machine learning using large datasets. **Ceramics International**, v. 47, n. 17, p. 23958-23972, 2021.

- LIBERA, Caio et al. 'Right to Be Forgotten': Analyzing the Impact of Forgetting Data Using-NN Algorithm in Data Stream Learning. In: **International Conference on Electronic Government.** Springer, Cham, 2022. p. 530-542.

- MASTELINI, Saulo Martiello et al. Machine learning unveils composition-property relationships in chalcogenide glasses. **Acta Materialia**, v. 240, p. 118302, 2022.

- RIBEIRO, Rita P. et al. Online Anomaly Explanation: A Case Study on Predictive Maintenance. In: **Machine Learning and Principles and Practice of Knowledge Discovery in Databases: International Workshops of ECML PKDD 2022, Grenoble, France,**

September 19–23, 2022, Proceedings, Part II. Cham: Springer Nature Switzerland, 2023. p. 383-399.


ICMC
USP
SÃO CARLOS