DATA STREAM CLASSIFICATION WITH MONDRIAN FOREST UNDER MEMORY CONSTRAINTS

Martin Khannouz

A thesis
in
The Department
of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of Doctor of Philosophy
Concordia University
Montréal, Québec, Canada

May 2023
© Martin Khannouz, 2023

Concordia University

School of Graduate Studies

This is to certify that the thesis prepared

By: **Martin Khannouz**

Entitled: **Data Stream Classification with Mondrian Forest Under Memory Constraints**

and submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Software Engineering)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining commitee:

______________________ Chair
Dr. Rolf Wuthrich

______________________ External Examiner
Dr. Katarina Grolinger

______________________ Examiner
Dr. Leila Kosseim

______________________ Examiner
Dr. Brigitte Jaumard

______________________ Examiner
Dr. Nizar Bouguila

______________________ Supervisor
Dr. Tristan Glatard

Approved ______________________
Dr. Leila Kosseim, Graduate Program Director

04/17/2023 ______________________

Dr. Mourad Debbabi, Dean
Gina Cody School of Engineering and Computer Science

Abstract

Data Stream Classification with Mondrian Forest Under Memory Constraints

Martin Khannouz, Ph.D.
Concordia University, 2023

Supervised learning algorithms generally assume the availability of enough memory to store data models during the training and test phases. However, this assumption is unrealistic when data comes in the form of infinite data streams, or when learning algorithms are deployed on devices with reduced amounts of memory. In this manuscript, we investigate the use of data stream classification methods under memory constraints. Our investigation consists of three steps: a benchmark of models, an update of a model, and an optimization of a trade-off. We evaluate data stream classification models with different criteria such as classification performance or resource usage. The benchmark reveals that the Mondrian forest, despite having state-of-the-art classification performance with unlimited memory, is impacted by a low memory limit. We then adapt the online Mondrian forest classification algorithm to work with memory constraints on data streams. In particular, we design five out-of-memory strategies to update Mondrian trees with new data points when the memory limit is reached. We evaluate our algorithms on a variety of real and simulated datasets, and we conclude with recommendations on their use in different situations: the Extend Node strategy appears as the best out-of-memory strategy in all configurations. We identify that the memory-constrained brings a trade-off between the Mondrian forest size and its tree depth. We design an adjusting algorithm to optimize the forest size to the data stream and the memory limit and we evaluate this algorithm on similar datasets. All our methods are implemented in the OrpailleCC open-source library and are ready to be used on embedded systems and connected objects. Overall, the contributions significantly improve the performance of the Mondrian forest under memory constraints.

Acknowledgments

I would like to thank my supervisor, Dr. Tristan Glatard, for his guidance. This thesis would not have been possible without his continued support and mentorship.

I would like to thank all my colleagues at the /bin lab for making this experience more enjoyable, especially to Valérie for her guidance and support during my time at Concordia.

I would like to thank my friends Ronny and Jeremy in France, who not only provided me with countless help but also motivated me to improve as a computer scientist and as a person.

Enfin, je remercie, en français, mes parents et mes beaux-parents pour leur soutien inconditionnel à tous mes projets.

Contribution of Authors

This thesis consists of four manuscripts. Author contributions are as follows.

Chapter 3: Martin Khannouz, Bo Li, and Tristan Glatard. OrpailleCC: a Library for Data Stream Analysis on Embedded Systems. *Journal of Open Source Software*, 4(39):1485, 2019.

I was responsible for software development, experimental design, model design, interpretation of results, and writing and editing the first draft of said manuscripts. Bo Li was a master's student and he aided with software development. Tristan Glatard provided supervision and aided with experimental and model design, and manuscript editing.

Chapter 4: Martin Khannouz and Tristan Glatard. A Benchmark of Data Stream Classification for Human Activity Recognition on Connected Objects. *Sensors*, 20(22):6486, 2020.

I was responsible for software development, experimental design, model design, interpretation of results, and writing and editing the first draft of said manuscripts. Tristan Glatard provided supervision and aided with experimental and model design, and manuscript editing.

Chapter 5: Martin Khannouz and Tristan Glatard. Mondrian Forest for Data Stream Classification Under Memory Constraints. *arXiv:2205.07871 [cs.LG]*, 2022.

I was responsible for software development, experimental design, model design, interpretation of results, and writing and editing the first draft of said manuscripts. Tristan Glatard provided supervision and aided with experimental and model design, and manuscript editing.

Chapter 6: Martin Khannouz and Tristan Glatard. Dynamic Ensemble Size Adjustment for Memory Constrained Mondrian Forest. In *2022 IEEE International Conference on Big Data (Big Data)*, pages 3358–3363, Osaka, Japan, dec 2022. IEEE Computer Society.

I was responsible for software development, experimental design, model design, interpretation of results, and writing and editing the first draft of said manuscripts. Tristan Glatard provided supervision and aided with experimental and model design, and manuscript editing.

Contents

List of Figures viii

List of Tables xi

Glossary 1

1 Introduction 1

2 Data Stream Processing 6

2.1 Classification and Data Stream Classification 7

2.2 Existing Data Stream Classification Models 9

2.2.1 Naive Bayes 9

2.2.2 k-nearest neighbors 9

2.2.3 Decision Tree 10

2.2.4 Ensemble classifier 12

2.2.5 Random Forest 13

2.2.6 Online and Data Stream Forests 14

2.2.7 Mondrian forest 16

3 OrpailleCC: a Library for Data Stream Analysis on Embedded Systems 18

4 A Benchmark of Data Stream Classification for Human Activity Recognition on Connected Objects 21

4.1 Introduction 22

4.2 Related Work 23

4.2.1 Comparisons of data stream classifiers 23

4.2.2 Offline and data stream classifiers for Human Activity Recognition . . . 24

4.3 Materials and Methods 25

4.3.1 Datasets 26
4.3.2 Algorithms and Implementation 28
4.3.3 Evaluation 32
4.3.4 Results Reproducibility 33
4.4 Results 34
4.4.1 Overall classification performance 34
4.4.2 Hoeffding Tree and Naïve Bayes 37
4.4.3 Mondrian forest 38
4.4.4 MCNN 38
4.4.5 Feedforward Neural Network 38
4.4.6 Power 39
4.4.7 Runtime 40
4.4.8 Memory 40
4.4.9 Hyperparameter tuning 40
4.5 Conclusion 42

5 Mondrian Forest for Data Stream Classification Under Memory Constraints 45
5.1 Introduction 46
5.2 Materials and Methods 47
5.2.1 Mondrian Forest 47
5.2.2 Mondrian Forest for Data Stream Classification 47
5.2.3 Out-of-memory Strategies in the Mondrian Tree 48
5.2.4 Concept Drift Adaptation for Mondrian Forest under Memory Constraint 51
5.2.5 Time Complexity 54
5.2.6 Node Boxes Analysis 55
5.2.7 Datasets 58
5.2.8 Evaluation Metric 59
5.3 Results 59
5.3.1 Baselines 60
5.3.2 Out-of-memory strategies 60
5.3.3 Concept Drift Adaptation for Mondrian Forest under Memory Constraint 62
5.3.4 Impact of the Memory Limit 63
5.4 Related Work 66
5.5 Conclusion 68

6 Dynamic Ensemble Size Adjustment for Memory Constrained Mondrian Forest **70**

6.1 Introduction 71

6.2 Materials and Methods 71

6.2.1 Mondrian Forest 72

6.2.2 Mondrian Forest for Data Stream Classification 72

6.2.3 Dynamic Tree Count Optimization 73

6.2.4 Comparison Test 74

6.2.5 Pre- and Postquential Statistics Computation 75

6.2.6 Tree Addition Method 76

6.2.7 Datasets 77

6.2.8 Evaluation Metric 78

6.3 Results 78

6.3.1 Optimal Forest Size 79

6.3.2 Tree Addition 79

6.3.3 Tree Removal 80

6.3.4 Comparison to Fixed Ensemble Size 81

6.4 Related Work 83

6.5 Conclusion 84

7 Conclusion **85**

7.1 Benchmark Contributions 85

7.2 Mondrian Forest Contributions 85

7.3 Other Contributions 86

7.4 Future Work 87

List of Figures

1.1 The power and energy consumption in connected devices (figure extracted from [25]). Yellow: Transmition of data, Blue: Reception of data, Red: Processing, Green: Sleep. 2
1.2 Architecture illustration of the Internet of Things (figure extracted from [82]). 2
2.1 Illustration of the Mondrian tree before (top) and after (bottom) of a branch out. The left side shows the data points in a two-feature space with the node's boxes and the splits. The right side shows the tree structure. The branch out is triggered by the purple data point. 17
4.1 F1 scores for the six datasets (average over 20 rep.). The horizontal dashed black line indicates the offline k-NN F1 score (the value of k was obtained by grid search in [2, 20]). The blue shades is the $\pm\sigma$ interval of the Mondrian forest classifier. 35
4.2 Power usage and Runtime (20 repetitions) for the Banos *et al* dataset. Results are similar across datasets. 39
4.3 Memory footprint of classifiers with the empty classifier as a baseline, measured on the Banos *et al* dataset. The memory footprint of the empty classifier is 3.44 MB. The baselines are the two Naïve Bayes from OrpailleCC and StreamDM-C++. Their respective memory footprints are 3.44 MB and 4.74 MB. 41
4.4 Error threshold tuning of MCNN with the first subject of Banos *et al* dataset. Error threshold in parenthesis. 42
4.5 Hyperparameters tuning for Mondrian with the first subject of Banos *et al* dataset. 43

5.1 A two-feature example of the split definition. In each scenario, rectangles represent leaf boxes, the triangle is the new data point, and the cross is the split helper. The green area indicates the range of values where the new split will be created. The split helper is arbitrarily placed to illustrate different situations. 54
5.2 Comparison of the out-of-memory strategies proposed in Section 5.2.3 for six datasets. 61
5.3 Comparison of trimming methods applied with the Extend Node out-of-memory strategy. 63
5.4 Comparison of tree leaf splitting methods combined with the Random trimming strategy. 65
5.5 Evaluation of the memory impact on the top out-of-memory strategy and the top trimming methods described in Section 5.2.3 and Section 5.2.4. The results are shown for 50 trees. 66
6.1 The impact of the ensemble size on the F1 score depending on the datasets and the memory limit. 79
6.2 The effectiveness of the adding methods and removing method compare to Fixed. 80
6.3 Comparison between component combinations and Fixed with the optimal tree count. Fixed is represented by a vertical dashed line. The value for the component combinations is percentage of the Fixed F1 score. 82

List of Tables

4.1 Activity merging in Recofit dataset (table extracted from [31]). 27
4.2 Hyperparameters used for the Mondrian forest. 29
4.3 Hyperparameters used for the MCNN. 30
4.4 Average F1 scores obtained on the last data point of the stream. 36
5.1 Summary of the node structure used in Algorithm 1. 50
5.2 Summary of the proposed out-of-memory strategies. `update_counters` and `update_box` refer to the functions in Algorithm 1. —: no-op. 51
5.3 ΔF1 score compared to Stopped Mondrian. Minimum, maximum, and average scores are computed across all tree numbers. 62
5.4 ΔF1 score compared to Stopped Mondrian for the trimming methods. Minimum, maximum, and average scores are computed across all tree numbers. . 64
6.1 The best-ranked component combinations out of the 24 combinations of Algorithm 4 across datasets and memory limits. The top lines rank better on average than the bottom lines. 83

Chapter 1

Introduction

The amount of connected devices is expected to double in the next five years to reach 75.4 billion in 2025 [77]. However, the current data processing architecture suffers limitations. Currently, this architecture is generally centralized, as illustrated in Figure 1.2, implying that data are produced on the device, then transferred to a centralized computer where they are stored and processed. This centralized approach suffers from a few drawbacks such as energy consumption and lack of privacy. Indeed, transferring the entirety of the data centrally requires a substantial amount of wireless communication, which importantly reduces battery life. Figure 1.1 shows that wireless communications represent 90% of the power consumption and 50% of energy usage [25, 4, 35].

Moreover, centralizing the data delegates data governance entirely to the central entity. This situation threatens privacy, for instance when political views, sexual orientation, or medical conditions can be extracted from the data [97]. Currently, the user does not know who is using their data, or when [27]. This is an issue since the data can be sold or stolen and used many years after their acquisition. Four approaches are used to enhance privacy: authentication and authorization, edge computing and plug-in architecture, data anonymizing and denaturing, digital forgetting, and data summarization [97].

A solution to privacy and battery issues is to decentralize data collection by processing data directly on a connected device using data stream algorithms. This solution is a combination of edge computing and data summarization. Indeed, data stream algorithms aggregate summaries of the data to central places, which limits the potential future use of the data and the amount of data transmitted through the wireless network [29, 99]. On the other hand, the decentralized architecture increases resource usage on the devices, in particular CPU and memory, which, in turn, may increase energy consumption. Additionally, the

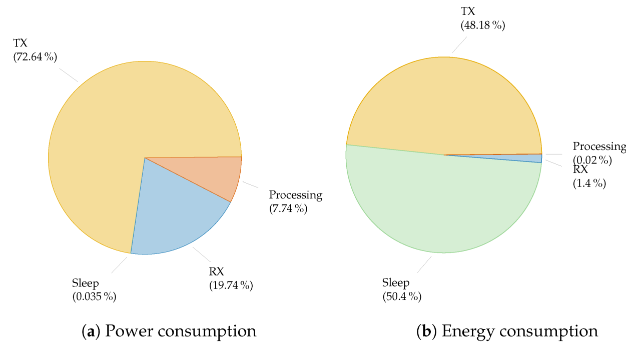


Figure 1.1: The power and energy consumption in connected devices (figure extracted from [25]). Yellow: Transmition of data, Blue: Reception of data, Red: Processing, Green: Sleep.

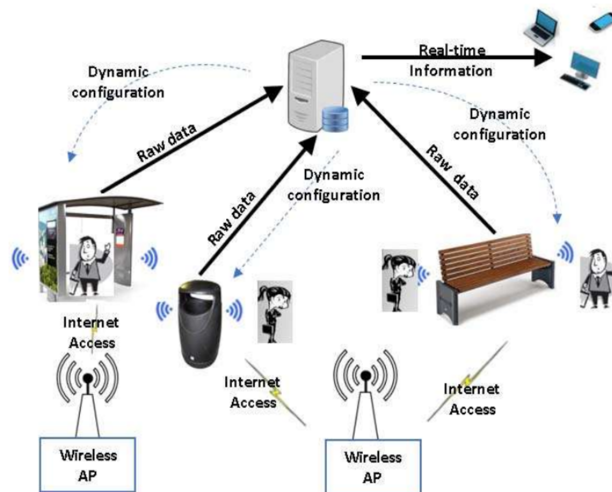


Figure 1.2: Architecture illustration of the Internet of Things (figure extracted from [82]).

resource limitation of connected objects (memory and processor) creates challenges.

A data stream algorithm is an approach to processing an infinite sequence of data points that arrive continuously over time. It is designed to handle high-velocity and high-volume data, and has constant space and time complexity, which makes it well-suited for processing data from connected objects. Data stream classification is a subclass of online classification, in which the model is updated with new data as it arrives [14, 88]. However, data stream classification is typically more constrained on the resources, the complexities, and on the volume of data than other online classification approaches. In this manuscript, we use the following definition: 1) Offline classification refers to situations where the entire dataset can be accessed without any restrictions 2) Online classification refers to scenarios where only one chunk of the dataset can be accessed at a time, but this chunk can be accessed again later 3) Data stream classification, on the other hand, refers to scenarios where the dataset is processed in a continuous streaming fashion. Many problems have been studied from the perspective of data stream processing such as sampling, counting, or learning [52, 83, 110, 59].

This manuscript focuses on supervised classification for data streams, a problem that has been targeted by many models. These models generally derive from existing non-data stream models and extend them to process infinite sequences of data points. There are individual models such as the Naïve Bayes or the Hoeffding Tree [33], and ensemble-based models such as the Adaptive-Size Hoeffding Tree [19], the Kappa Updated Ensemble [26], or the Mondrian forest [71]. Some models are hybrid such as Fast-Slow Classifier [78] that combines a Hoeffding Tree and an ensemble.

Among them, the Mondrian forest is an ensemble of randomized decision trees [71]. Similarly to Random Forests [24], these trees involve randomness to grow differently from the same data, limiting the ensemble from overfitting. The performances reported in [71] show that the classification performance of the Mondrian forest is similar to the offline Random Forests and Online Random Forests [93], the straightforward extension of Random Forests for online learning. The Mondrian forest has the particularity of not using labels during training, but rather selecting the splits based on data point density only. Even though the labels are required for the prediction, they can be delayed during the tree-building phase, a situation that occurs for some data streams [50, 48].

Most ensemble-based models presented before assume the existence of enough memory to train and update to new data [78, 71, 26, 93]. These models usually crash if the model size exceeds the available memory [12, 54]. The assumption of unlimited memory clearly does not hold when the architecture is decentralized and the learning happens on the devices. This

manuscript focuses on memory-constrained Mondrian forests for data streams. A memory-constrained model is a model where the memory footprint should remain under a user-defined limit. In general, the model classification performance is directly linked to the memory limit and the model should, therefore, aim at using all the memory available. The overall goal of this thesis is to develop a Mondrian forest algorithm suitable for data stream classification with memory constraints.

This thesis is article-based and this manuscript compiles the publications associated with the thesis. Chapter 2 provides the background needed to understand the contributions. It introduces data stream, data stream classification, and the problems posed by processing streaming data. The chapter also describes existing data stream classification algorithms.

Chapter 3 describes OrpailleCC and was published in [63]. OrpailleCC is an open-source collection of various data stream algorithms including supervised learning algorithms. The OrpailleCC paper is published in the Journal of Open Source Software which enforces the best software practices by reviewing the source code of the library linked to the papers. In particular, the journal publishes papers where the linked source code is released with extensive documentation and unit tests. In OrpailleCC, the algorithms are implemented with the deployment of connected objects in mind. In particular, the learning algorithms have a constant memory footprint.

Chapter 4 is a benchmark of data stream classifiers published in [60]. The benchmark tests classifiers implemented in OrpailleCC, including the Mondrian forest. In contrast with the original Mondrian forest implementation, the OrpailleCC implementation includes a memory limit set by the user. The benchmark evaluates the model F1-scores and resource usage such as memory or energy consumption. The benchmark concludes that the implementation of the Mondrian forest maintains an overall good classification score, but is sensitive to the amount of available memory. Indeed, when less memory is available the classification performances worsen. The remaining chapters focus on the Mondrian forest with a memory constraint and aim at making it memory efficient given a user-defined memory limit. This chapter focuses on Human Activity Recognition because prior to this benchmark, a collaboration was established with Motsai company, which was interested in this application. However, the algorithmic contributions of this thesis have broader applications.

Chapter 5 proposes and evaluates 14 designs for the memory-constrained streaming Mondrian forest and is published in [62]. In particular, it defines and tests 5 ways to update the Mondrian forest while complying with the memory limit. Additionally, it defines and evaluates 9 mechanisms to handle concept drifts. The chapter identifies the Mondrian forest

design that is best suited for data stream learning with a memory constraint. This design improves the Mondrian forest F1 score by 0.27 on average compared to the naïve default strategy. The experimental results reported in this chapter exhibit the impact of the ensemble size on the classification performance of the Mondrian forest. In particular, the results denote the existence of an optimal ensemble size beyond which the performance of the forest decreases. This is usually not the case in tree-based ensembles, such as Random Forest [24], when trained without memory constraint.

Chapter 6 explores how the ensemble size of the Mondrian forest can be dynamically adjusted to reach its optimal size and it is published in [61]. Our results highlight a trade-off between tree depth and tree count. Indeed, due to the memory constraint, having more trees in the forest leads to smaller trees that underfit the data. This trade-off makes the Mondrian forest either overfit when there are too few trees, or underfit when there are too many trees. Then we propose a method based on overfitting detection to guide the growth of the ensemble within the memory limit. The chapter concludes that the method works even though the hyper-parameters still need tuning. Using our method, the model can achieve up to 95% of an optimally-sized Mondrian forest.

Overall, the main contribution of this thesis are 1) a benchmark of data stream classifiers; 2) the adaptation of the Mondrian forest to data stream and memory constraints; 3) a mechanism to optimize the forest size under memory constraint; 4) an open-source implementation of the memory-constrained Mondrian forest.

Chapter 2

Data Stream Processing

Data Stream Algorithms are designed to process an infinite sequence of items that may be read only once [90, 10, 47]. A data stream is considered endless because the amount of data that will be produced is unknown. Therefore, the size of the sequence is not available and must be considered infinite. An item in the data stream must be processed a constant number of times, usually only once, because multiple examinations require storing the item for a longer period of time. As a data stream is infinite, examining items an unknown number of times requires storing the entire data stream which is not possible. Additionally, examining one item has to be done with a constant complexity regarding the number of items already processed. This limitation is related to the infinity of the stream because the time to process one item will increase too much and the algorithm will not be usable after the processing time has exceeded the period between two consecutive elements. Finally, the distribution of values in a data stream might be changing over time and the most recent data are often more relevant than the older ones. This behavior is called concept drift.

Data streams appear in many situations that involve connected devices. A camera produces a series of pictures, a connected wristband senses the evolution of heartbeat through time, and a refrigerator emits every time an item is used. All previous examples stream a series of events, possibly endless, from which data must extract on the fly.

When processing these infinite sequences with limited examinations, key information should be extracted as soon as the item is available even though the entire sequence is not available, rather than waiting for all items. Delaying the extraction may result in the loss of valuable information. However, due to the lack of data and the constant time complexity, an approximate answer is often acceptable. Therefore, most data stream algorithms are stochastic and include parameters to tune the error margin. For instance, in the camera

example mentioned above, detecting a particular object in an image can be achieved with a given confidence level and this level is adjusted depending on the user's need.

Data stream algorithms cover a wide range of problems. The work in [59] gathers a list of those problems and associates papers with each of them. These papers explore data stream methods as well as online methods. Online algorithms differ from data stream algorithms by the availability of the data. In online situations, the data are already available and the dataset size is known. Nevertheless, the dataset is too large to fit in memory, thus it must be processed in sequence (preferably once). In the rest of this manuscript, we will focus on data stream classification.

2.1 Classification and Data Stream Classification

Supervised classification is a machine learning problem in which a model has to accurately predict the label (or category) of an unlabeled data point. The model is trained on labeled data points available in a dataset. A dataset is a set of data points where each point corresponds to one element. A point is described by its features (for instance, length, width, or any information that characterizes the point) and all points from a dataset have the same number of features.

Offline classification is when the dataset is available without limits. The dataset is split in two: the training set (labeled points) is used to build a model then a test set (unlabeled points) is used to evaluate it. Online classification is when the training set is only available in sequential order and each data point is used to update the model. However, the model can be built with multiple passes through the training set.

Data stream classification introduces further challenges to online classification because the dataset is an infinite sequence [43]. The length of the stream is unknown, so it cannot be used to set variables or adjust probabilities. Given that the stream may be infinite, storing the stream is not feasible, and the model must maintain constant time and memory complexities. The class distribution is unknown, and there may be a class imbalance that fluctuates throughout the stream.

Moreover, data streams are sensitive to concept and data drifts, a common challenge in data stream classification where the underlying patterns in the data change over time. It can happen for various reasons, including changes in user preferences, external factors, or seasonality. Drift creates a significant challenge for classifiers since they learn patterns from historical data, and when these patterns change, the models may no longer be accurate.

Thus, detecting and adapting to drift is essential to ensure that the machine learning models remain effective in dynamic environments. It is a difficult problem that is further complicated by memory constraints, as drift detection and adaptation require memory.

Concept drift and data drift are two types of changes that can occur in a data stream. Data drift refers to a change in the distribution of input features over time. This can be caused by various factors, such as changes in the data collection process, changes in the underlying system generating the data, or data anomalies. Concept drift, on the other hand, refers to a change in the underlying relationship between the input features and the target variable. This means that the statistical properties of the target variable can change, even if the input features remain the same. For example, a feature that was once important for predicting a target variable may become less relevant over time, while other features may become more important.

Even though the environment is dynamic, in this manuscript we assume a fixed number of features and labels throughout the data stream classification process.

These obstacles related to data stream learning disrupt the offline pipeline where datasets are split in training set and test set. Instead, the metric used is the prequential evaluation [30] also known as the interleaved test-then-train. With this metric, each new data point from the stream is used to first test the model, then train the model. Additionnaly, the prequential evaluation can be combined with a fading factor that gives more weight to the recent data points, and therefore limiting the impact of old misclassifications [43].

There are two approaches to adapt existing models to data streams: batch learning and incremental learning. Batch learning collects data points in a sample large enough to be representative of the stream, then trains a new offline model with that batch. Batch learning has the advantage of re-using the existing models without much modification. However, it is more resource-intensive because it re-trains a new model and it has to store enough data points for the offline model.

The incremental learning approach updates the model with each new data point without the need to store many of them. This approach is less resource-intensive, but it requires to design new models.

When training a model, there are a few general problems that raises: bias, variance, underfitting, and overfitting. The bias is the error on the training set. The variance is the error on the test set. The model underfits when it has a high bias because it misses relevant relations between the features and the labels. When a model has a low bias and a high variance, the model overfits, because it has taken into account random noise in the training

data.

2.2 Existing Data Stream Classification Models

In this section, we present various classification models and their data stream adaptations. We first describe the offline models, then we present adaptations of these models to data streams. The models can use both incremental and batch learning.

2.2.1 Naive Bayes

The Naive Bayes classifier [72] is a probabilistic classifier based on Bayes Theorem. It aims at finding the most likely class of a record based on the probability of the attributes. To proceed, the classifier needs the dataset with all possible classes. The Naive Bayes algorithm compiles the dataset into a table of occurrences and it uses this table to compute the probabilities to belong to each class given a feature value. To classify a new record, probabilities of the features are combined for each possible class.

This approach is datastream-friendly for many reasons. Adding a new record to the model has a constant complexity because at most one row of counters needs to be incremented. The size of the model is also constant and very small (number of classes $\times$ number of features). The table of occurrences is only limited by the bit size of its counters and does not store the data points. Because of the simplicity of this method, Naive Bayes can be employed by bigger classification models for smaller subsets. Finally, the Naïve Bayes can be modified to focus on the most recent data, which helps recover from concept drifts [109].

Even though the Naïve Bayes classifier is a popular and straightforward choice, it has a few drawbacks. The model assumes that features are independent, which is not always the case in real life. Additionally, continuous features need to be discretized. Furthermore, the model does not work with missing data and needs alternative methods such as *smoothing* to behave properly. Finally, the performance of the Naïve Bayes classifier can be poor compared to other classifiers [31].

2.2.2 k-nearest neighbors

The k-nearest neighbors (kNN) [6] classifier uses the neighbors accumulated during the training phase to classify a new element. Given an element e and a constant k, the algorithm retrieves the class of the k-nearest neighbors of e in the training set. The predicted class for

e is the majority class amongst the k neighbors. Even though the Euclidean distance is the most popular one, other distances can be used, such as Manhattan, Minkowsky, Chebychev or Camberra [67].

The kNN algorithm has a major inconvenience for online and datastreams applications: it uses the whole training dataset to classify a data point. This behavior contradicts two principles of data stream learning: data should be processed once and the stream cannot be stored. However, using a sliding window to store the last records of the dataset makes it viable for online applications. In addition, the sliding window allows kNN to better catch concept drifts. This side effect appears because the dataset for kNN does not contain the oldest elements, thus the older concepts. The work in [2] uses a biased reservoir sampling method to build a sample before applying kNN. Thus, new points are classified using their k-nearest neighbors from the sample. Results show that the kNN over the biased window performs better than the kNN over an unbiased sample.

Micro-Clusters Nearest Neighbors [103] (MC-NN) is a kNN adaptation to data streams where elements are aggregated in centroids. MC-NN compresses the data points and diminishes the amount of comparisons to execute in the classification phase. To adjust to concept drift, MC-NN splits the centroids when they reach a given amount of misclassification.

ProtoNN [51] is a kNN-based model that performs a low-dimension projection of the features to increase accuracy and improve its memory footprint. The model also compresses the training set into a fixed amount of clusters. ProtoNN claims to remain below 2KB while retaining high accuracy. However, this model does not apply to evolving data streams because its low-dimension projection and its structures are pre-trained based on existing data, thus, adjusting them would require more time and memory.

2.2.3 Decision Tree

A decision tree [89] is a type of classifier structured as a tree where nodes represent classification criteria. Each internal tree node contains a criterion on the features to split the dataset between its child nodes, and tree leaves contain records from the dataset to finalize the classification.

During the training phase, all records are passed down the tree following the tests in internal nodes until they reach a leaf. Then a metric is computed to decide whether the leaf needs to be split and which criterion should be used for the split. The most common metrics are entropy [98] through the information gain and the Gini index [46].

During the test phase, records pass through the tree until they reach a leaf. In each

internal node, the features are tested with the tests selected during the training phase. The test result redirects the record toward one of the child nodes. Once the record reaches a leaf, it is classified using the training element that also attained this leaf.

Decision trees are not suitable for data stream learning. Indeed, to compute the information gain of a possible split, the tree needs to store all records of the leaf. Regarding the root, the complete dataset is required to compute the first split, which violates one of the principle of data stream learning.

To bypass this limitation, incremental trees were developed. An incremental tree is a decision tree to which training records are given one by one. The main difficulty in incremental trees is to decide when the split should be done and when to discard a record that is not needed anymore.

The Hoeffding Tree [33] tackles these issues in two ways. First, it stores feature counts of every record in each leaf. This operation removes the need to store the data points. Second, it uses the Hoeffding bound equation to decide when the difference between the two best criteria is high enough in order to proceed to a split. Similarly, the McDiarmid Tree [92] is a decision tree where the McDiarmid bound replaces the Hoeffding inequality.

In order to improve the Hoeffding Tree performance, the Hoeffding Option Tree (HOT) uses option nodes [66] to allow records to be sorted in multiple leaves. Then a weighted vote algorithm combines the classification of all leaves. Even though this approach is similar to ensemble methods (Section 2.2.4), it uses much less memory while preserving the performances. The HOT also describes pruning methods to reduce further the memory footprint to the same point as the Hoeffding Tree at a small cost in accuracy.

The Hoeffding Adaptive Tree [16] (HAT) is a variant of the Hoeffding Tree developed to deal with concept drifts. In addition to statistics on leaves for the splits, it also keeps information at the node level, to detect drift. If a drift is detected, the algorithm starts growing an alternate branch at that node called a ghost branch. If the ghost branch obtains better results than the original one, the original branch is replaced by the new one. The HAT proposes two methods to detect concept drifts: one based on a sliding window and a second one based on the Adaptive Windowing detector [5].

Adaptive Windowing (ADWIN) is a change detection algorithm used in data stream mining. It dynamically adjusts a sliding window over a stream of data, constantly monitoring the data distribution in the window. If a significant change is detected, the window size is adjusted to keep more recent data and exclude older data. ADWIN is designed to adapt to changes in the data distribution quickly and with logarithmic space complexity.

Overall, to propose a new data stream decision tree, a developer has to fix two issues. The new algorithm has to be built iteratively, thus the tree needs to know when it should process to a split, with the most common method being the Hoeffding bound. The tree also needs to adapt to concept drifts which imply reshaping itself. Additionally, the tree may need to detect concept drift in order to adapt properly, but it is not mandatory.

2.2.4 Ensemble classifier

Ensemble classifiers are classifiers that use a group of sub-classifiers (or weak classifiers) to make their prediction [34]. When an ensemble classifier makes a prediction, it runs all its sub-classifier, then merges the result using a voting system, for instance, the majority class or a consensus vote.

Boosting is an ensemble classifier where sub-classifiers are trained sequentially and each sub-classifier is trained with the errors of the previous one in mind. The most famous boosting algorithm is Adaboost [42]. At the first iteration, all data points have a weight of one, then weights of misclassified records in the previous iteration will represent half the weight of the dataset. Therefore, the following sub-classifier will focus on the misclassified points.

Bagging [23] is another ensemble classifier where sub-classifiers are trained concurrently over slightly different datasets called bags. A bag is a sample with replacement of the dataset. When the size of the bag is equal to the size of the original dataset, the bag is called a bootstrap sample. This ensemble classifier provides several advantages: it is straightforward to parallelize and it limits overfitting. Indeed, since all sub-classifiers are independent, they can train in parallel. Also, since the sub-classifiers are not built with the same training set, the ensemble is less likely to focus on random noise in the data.

Bagging introduces a new concept to measure performance: the out-of-bag error (OOB) [56]. When a bag from the training set is assembled, some records are left out of the bag. It happens because of the replacement or because the size of the bag is smaller than the training set. The OOB error of a sub-classifier is the average prediction error on the records out of its bag. The OOB error of the ensemble is the average error of prediction for each training record with each sub-classifier when the record does not belong to its bag.

Ozabag and Ozaboost [84] adapt bagging and boosting to online applications. The online bagging assumes the dataset length tends to $+\infty$ and thus, is able to compute the probability of one element to appear a certain amount of times k. This number k follows a Poisson law. Therefore, it allows the online bagging to generate a number k following this law for each

record and then insert the record k times. This online approach performs as well as the offline bagging.

On the other hand, online boosting proposed in [84], slightly twists the original approach. Instead of training one classifier with the whole dataset and then adjusting the weights, it trains each sub-classifier with the current record and adjusts the weight based on the classification result of the sub-classifier. This online boosting does not match the performance of the offline version even though it remains close to it.

Fast and Slow Classifier [78] (FSC) is a hybrid method to cope with data stream learning that involves incremental learning with a single decision tree and batch learning with an ensemble. Indeed, FSC combines a batch learning algorithm (M_s) and a stream learning algorithm (M_f) to provide the best prediction. FSC uses an Hoeffding Tree as M_f and an eXtreme Gradient Boosting as M_s. M_f is continuously trained with each new item while M_s is only trained on a sliding window. When M_s is not trained yet, M_f is used to classify. When both M_s and M_f are available, the one with the highest accuracy is used. The error of each model is continuously updated with the prequential error [30]. ADWIN [5] is used to detect concept drift. When a drift is detected, the sliding window is split into two sets (training and testing) and new models of M_s are trained based on the training set. FSC uses four strategies to build a new model. The first uses the entire training set unchanged. The second strategy trains the model over a probabilistic adaptive window of the training set. The third approach trains the model on a reservoir sample of the training set. The last strategy uses a weighted reservoir sampling algorithm. Once the new models are trained their performances are compared on the testing set. The original M_s is also tested in case it remains more accurate than the new models. Finally, the most efficient model will replace M_s. In most cases, the hybrid model outperforms its two components, M_s and M_f.

2.2.5 Random Forest

A Random Forest [24] is an ensemble of decision trees. Given a training dataset $\mathcal{D}$ where each record is represented by k features, each tree t_j of the forest is trained with a bag $\mathcal{D}_j$. As described before, $\mathcal{D}_j$ is a sample with replacement from the original dataset $\mathcal{D}$. When the tree grows, each leaf considers l features randomly picked to select a split, with $l \ll k$. Thus, a leaf only considers a subset of the features to select a split. The original paper [24] also considered recombining features with linear combinations, which gives slightly better results. To select the best splits in trees, the Random Forest relies on the Gini index [46] rather than the information gain.

To predict the class of an unlabelled record, the algorithm passes this record through all trees and returns the majority class predicted by them. This algorithm benefits from the bagging method by not overfitting the data and by being straightforward to parallelize.

The forest performances converge with the number of trees and using a larger l to select the split enhances accuracy [24].

Since 2001, Random Forests have been widely explored. A survey found that ReliefF measurement gives better performance than the Gini index [69]. However, combining various information measures (Gini, Gain ratio, ReliefF, MDL) does not improve the result. An empirical study showed that a forest with 100 trees and $l = \log(feature + 1)$ offers the best performance on imbalanced datasets [64]. It also shows that a Random Forest with these parameters performed better than a baseline composed of kNN, C4.5 decision tree, Naïve Bayes, and SVM.

While the original Random Forest uses the majority vote to aggregate the classification of the trees, the survey in [69] describes various alternatives. All of them weigh the vote of each tree based on its characteristics. Tree accuracy is one of them. Dynamic Integration aims at replacing the majority vote with a dynamic voting system [104]. Each variant of the Dynamic Integration stores the result of all trees for all records in the training set. Then, kNN finds similar instances of the unlabelled data point for each tree. Finally, these instances are used to predict the error of each tree for the new data point. Depending on the variant, the system uses these error predictions to weigh trees, discard them, or both.

Another improvement in Random Forests is to find a smaller forest that achieves the same accuracy. Reducing the forest size improves the memory footprint and decreases the amount of time to train the forest. However, most methods tend to overproduce trees and then shrink the forest by removing trees or combining them [69]. If these approaches provide smaller forests, they still have the same peak memory and use the same time to produce.

Finally, Random Forests have been adapted to various types of situations. Initially designed to classify and make regressions [24], some Random Forests version estimate quantiles, applies clustering algorithms and predicts ranking [15].

2.2.6 Online and Data Stream Forests

Online and data stream Random Forests raise new problems. Building bags out of a stream is not straightforward and choosing the right manner of adapting to concept drift is subject to debate. Indeed, Random Forests have at least two options to handle drifts: changing trees or using incremental trees designed for data streams. Furthermore, data

stream forests, as classification algorithms, have to handle non-uniformly distributed labels over the sequence. In addition, the classification problem faces constraints related to data streams as explained in the beginning of this chapter. Data should be read once, the learning time complexity has to be constant for each record, and the sequence cannot be stored.

The Streaming Random Forest [1] is a method that combines Hoeffding Trees [33] to form a forest. Trees are trained with n records and tested with 10% of this amount. As long as the tree error remains too high, the training continues with a decreasing number of records ($\frac{n}{2}$, $\frac{n}{4}$, $\frac{n}{4}$, etc $\cdots$). During the prediction phase, the tree error is used to weigh the tree prediction and thus, decrease the influence of the tree with a higher error. To detect concept drift, the method compares the entropy between an older window of the stream and the current sliding window. Periodically, the method checks for concept drift intensity and changes between 25% and 100% trees starting with the trees with the largest error. A minimum of 25% of the forest is always changed to ensure that even small concept drifts are caught.

In the same vein, the Adaptive-Size Hoeffding Tree [19] combines Hoeffding Trees with a different size limit for each. When such a tree reaches its limit, it restarts from the last added node and deletes the other nodes. With this mechanism, trees with smaller limit will adapt faster to concept drift.

Ultra Fast Forest Tree [44] (UFFT) is an online Random Forest based on an online decision tree method for binary classification. The tree is designed to work with online data. Even though the tree is limited to a two-class classification, the paper suggests using a forest of these trees to handle multi-class classification. In addition to the trees, the UFFT uses a sliding window to initialize statistics in leaf nodes after a split. Each considered split is formed with an attribute and a value such that $attribute_j < value$. The value is selected with an analytical method [76] that solely requires the average and the variance: two statistics computable in an online fashion. However, this method assumes that attributes are uniformly distributed for each class. To turn a leaf node into an internal node, the algorithm waits for two events. A positive value from the information gain function toward the best split and having seen *enough* new data points so it is statistically supported by the Hoeffding bound.

To be classified, new records are passed through the tree until they reach a leaf, then a Naïve Bayes algorithm is applied to finalize the classification. With multi-class problems, the answers of each tree are a probability distribution for each class, then these probabilities are aggregated using the *sum rule* [65] and the most probable class is returned.

The Online Random Forest [93] (ORF) uses the Ozabag to build bags out of the data

sequence. The ORF also includes a mechanism to remove old trees based on their out-of-bag error (OOBE). The algorithm randomly picks a tree with an OOBE higher than a threshold and replaces it with a probability equal to the OOBE.

Instead of throwing a tree, the Adaptive Random Forests [48] proposes an alternative mechanism called the background tree. In this method, two thresholds of concept drift detection are set: *warning* and *replace*. When the *warning* threshold is crossed for a specific tree t, a background tree associated with t starts to grow. Both t and its background tree use the same records. When t crosses the *replace* threshold, it is replaced by its background tree. This system with two levels of concept drift is used to avoid using a brand-new tree in the forest. Instead, the tree had time to train on the recent element of the stream.

Even though the tree error is not used to discard trees in Adaptive Random Forests [48], the tree error is used in a weighted vote for prediction. Additionally, the Adaptive Random Forests carries little difference compared to the Online Random Forest. It uses Hoeffding trees instead of regular CART trees and it uses ADWIN to detect concept drift instead of out-of-bag errors.

2.2.7 Mondrian forest

The Mondrian forest is an ensemble of Mondrian trees [71] and it is the main focus of this manuscript. The Mondrian tree is a decision tree that involves a lot of randomnesses to build different trees given the same data points. The performances reported in [71] show that the Mondrian forest has competitive performances similar to the Random Forest [24], the Extremely Randomized trees [45], and the Online Random Forests [93].

In a Mondrian tree, each node is made of label counters and a feature box, thus keeping the minimum and maximum range of each feature. At any node, all the points counted at that node fit inside the node's feature box. Additionally, the node's box fits inside its parent's box. To grow the tree, the data points are passed down the tree either one by one or in batches. If a data point is sorted to a node (internal or leaf) but the point falls outside of the node's box, then the node may trigger a branch out, as illustrated in Figure 2.1. Branching out is a way to introduce a new leaf at any place in the tree. In Figure 2.1 (top), a new data point (in purple on the left) is sorted into leaf C but falls outside of C's box. Figure 2.1 (bottom) shows the result of a branch out with two new nodes: F and F. F is a node that will hold the purple data point, and E is the parent of F and C. The new nodes are introduced without modifying existing branches or altering their counters. The probability to trigger a branch out depends on how far the data point is from the node's box. If a branch

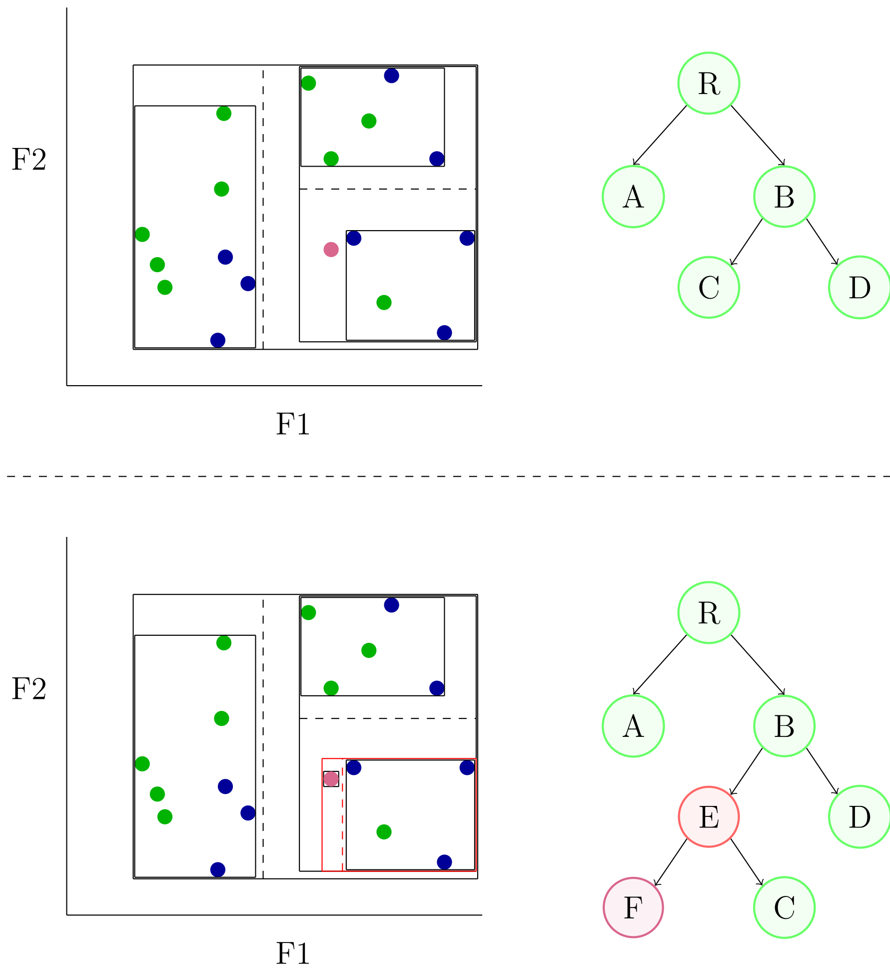


Figure 2.1: Illustration of the Mondrian tree before (top) and after (bottom) of a branch out. The left side shows the data points in a two-feature space with the node's boxes and the splits. The right side shows the tree structure. The branch out is triggered by the purple data point.

out is not triggered for a point outside of a node's box, the node's box is extended to fit that point.

For prediction, the unlabelled data point is passed down each tree and they give an array of probability. The forest averages the prediction of its trees and predicts the label with the highest average probability.

The model proposed in [71] and its implementation made in [12] is an online algorithm that requires a few adjustments to properly work for data streams. A detailed description of the Mondrian forest is done in Chapter 5.2.2, where we discuss different ways of adapting the Mondrian forest to data streams.

Chapter 3

OrpailleCC: a Library for Data Stream Analysis on Embedded Systems

Published as: Martin Khannouz, Bo Li, and Tristan Glatard. OrpailleCC: a Library for Data Stream Analysis on Embedded Systems. *Journal of Open Source Software*, 4(39):1485, 2019.

The Journal of Open Source Software (JOSS), where OrpailleCC is published, is an open-access journal that publishes research software packages with high-quality documentation, peer-reviewed and well-tested source code. JOSS is designed to provide a publishing venue for researchers who develop and maintain open-source software that has been evaluated and tested for scientific research. JOSS operates on a model of community-driven review, where reviewers openly review the software package and make suggestions for improvements.

The guidelines of JOSS require software packages to be well-documented, well-structured, and to have an open-source license. The software should also be tested and used in a scientific context. JOSS encourages the use of community-based review, which helps to ensure the quality of the software and provides an opportunity for developers to receive feedback on their work. The papers submitted alongside the package are generally between 250 and 1000 words, which is why this chapter is relatively short.

These guidelines are relevant for the scientific community because they promote the use and development of open-source software that can be used and modified by researchers worldwide. By requiring high-quality documentation, JOSS makes it easier for researchers to understand and use the software, which can increase the reproducibility and transparency of

research. Finally, the community-driven review process ensures that the software is rigorously evaluated and tested, which can increase confidence in the quality and reliability of the software.

The remainder of this chapter present the short paper published in JOSS to describe the contribution.

The Internet of Things could benefit in several ways from mining data streams on connected objects rather than in the cloud. In particular, limiting network communication with cloud services would improve user privacy and reduce energy consumption in connected devices. Besides, applications could leverage the computing power of connected objects for improved scalability.

OrpailleCC provides a consistent collection of data stream algorithms developed to be deployed on embedded devices. Its main objective is to support research on data stream mining for connected objects, by facilitating the comparison and benchmarking of algorithms in a consistent framework. It also enables programmers of embedded systems to use out-of-the-box algorithms with an efficient implementation. To the best of our knowledge, existing libraries of stream mining algorithms cannot be used on connected objects due to their resource consumption or assumptions about the target system (e.g., existence of a 'malloc' function). Nevertheless, for more powerful devices such as desktop computers, Java frameworks such as Massive Online Analysis [18] and WEKA [53] achieve similar goals as OrpailleCC.

OrpailleCC targets the classes of problems discussed by [59], in particular Sampling and Filtering. Sampling covers algorithms that build a representative sample of a data stream. OrpailleCC implements the reservoir sampling [108] and one variant, the chained reservoir sampling [11]. Filtering algorithms remove the stream elements that do not belong to a specific set. OrpailleCC implements the Bloom Filter [22] and the Cuckoo Filter [39], two well-tested algorithms that address this problem.

In addition to Sampling and Filtering, OrpailleCC provides algorithms for stream Classification and for stream Compression. The Micro-Cluster Nearest Neighbour algorithm [103] is based on the k-nearest neighbor to classify a data stream while detecting concept drifts. The Lightweight Temporal Compression [96] and a multi-dimensional variant [74] are two methods to compress data streams.

All implementations rely as little as possible on functions provided by the operating system, for instance 'malloc', since such functions are typically not available on embedded

systems. When algorithms cannot be implemented without such functions, the library uses template parameters to request the required functions from the user. All algorithms are developed for FreeRTOS [8], a free real-time operating system used in embedded systems, but they should work on any micro-controller with a C++11 compiler. The C++11 programming language was chosen for its performance as well as its popularity in the field. All methods are tested and tests are run through Travis-CI.

In the future, we plan to extend the library with other reliable algorithms to widely cover as many common problems as possible. We also plan to use it as a basis to design new stream classification methods. External contributions are, of course, most welcome.

Chapter 4

A Benchmark of Data Stream Classification for Human Activity Recognition on Connected Objects

Published as: Martin Khannouz and Tristan Glatard. A Benchmark of Data Stream Classification for Human Activity Recognition on Connected Objects. *Sensors*, 20(22):6486, 2020.

This paper evaluates data stream classifiers from the perspective of connected devices, focusing on the use case of Human Activity Recognition. We measure both classification performance and resource consumption (runtime, memory, and power) of five usual stream classification algorithms, implemented in a consistent library, and applied to two real human activity datasets and to three synthetic datasets. Regarding classification performance, results show an overall superiority of the Hoeffding Tree, the Mondrian forest, and the Naïve Bayes classifiers over the Feedforward Neural Network and the Micro Cluster Nearest Neighbor classifiers on 4 datasets out of 6, including the real ones. In addition, the Hoeffding Tree, and to some extent the Micro Cluster Nearest Neighbor, are the only classifiers that can recover from a concept drift. Overall, the three leading classifiers still perform substantially lower than an offline classifier on the real datasets. Regarding resource consumption, the Hoeffding Tree and the Mondrian forest are the most memory intensive and have the longest runtime, however, no difference in power consumption is found between classifiers. We conclude that stream learning for Human Activity Recognition on connected objects is challenged by two factors which could lead to interesting future work: a high memory

consumption and low F1 scores overall.

4.1 Introduction

Internet of Things applications may adopt a centralized model, where connected objects transfer data to servers with adequate computing capabilities, or a decentralized model, where data is analyzed directly on the connected objects or on nearby devices. While the decentralized model limits network transmission, increases battery life [4, 35], and reduces data privacy risks, it also raises important processing challenges due to the modest computing capacity of connected objects. Indeed, it is not uncommon for wearable devices and other smart objects to include a processing memory of less than 100 KB, little to no storage memory, a slow CPU, and no operating system. With multiple sensors producing data at a high frequency, typically 50 Hz to 800 Hz, processing speed and memory consumption become critical properties of data analyses.

Data stream processing algorithms are precisely designed to analyze virtually infinite sequences of data elements with reduced amounts of working memory. Several classes of stream processing algorithms were developed in the past decades, such as filtering, counting, or sampling algorithms [59]. These algorithms must follow multiple constraints such as a constant processing time per data element, or a constant space complexity [43]. Our study focuses on supervised classification, a key component of contemporary data models.

We evaluate supervised data stream classifiers from the point of view of connected objects, with a particular focus on Human Activity Recognition (Human Activity Recognition). The main motivating use case is that of wearable sensors measuring 3D acceleration and orientation at different locations on the human body, from which activities such as gym exercises have to be predicted. A previously untrained supervised classifier is deployed directly on the wearables or on a nearby object, perhaps a watch, and aggregates the data, learns a data model, predicts the current activity, and episodically receives true labels from the human subject. Our main question is to determine whether on-chip classification is feasible in this context.

We evaluate existing classifiers from the complementary angles of (1) classification performance, including in the presence of concept drift, and (2) resource consumption, including memory usage and classification time per element (latency). We consider six datasets in our benchmark, including three that are derived from the two most popular open datasets used for Human Activity Recognition, and three simulated datasets.

Compared to the previous works reviewed in Section 4.2, the contributions of our paper are the following:

- We compare the most popular data stream classifiers on the specific case of Human Activity Recognition;
- We provide quantitative measurements of memory and power consumption, as well as runtime;
- We implement data stream classifiers in a consistent software library meant for deployment on embedded systems.

The subsequent sections present the materials, methods, and results of our benchmark.

4.2 Related Work

To the best of our knowledge, no previous study focused on the comparison of data stream classifiers for Human Activity Recognition in the context of limited memory and available runtime that characterizes connected objects.

4.2.1 Comparisons of data stream classifiers

Data stream classifiers were compared mostly using synthetic datasets or real but general-purpose ones (Electrical, CoverType, Poker), which is not representative of our use case. In addition, memory and runtime usage are rarely reported, with the notable exception of [21].

The work in [86] reviews an extensive list of classifiers for data streams, comparing the Hoeffding Tree, the Naïve Bayes, and the k-nearest neighbor (k-NN) online classifiers. The paper reports an accuracy of 92 for online k-NN, 80 for the Hoeffding Tree, and 60 for Naïve Bayes. The study is limited to a single dataset (CoverType).

The work in [58] compares four classifiers (Bayesnet, Hoeffding Tree, Naïve Bayes, and Decision Stump) using synthetic datasets. It reports a similar accuracy of 90 for the Bayesnet, the Hoeffding Tree, and Naïve Bayes classifiers, while the Decision Stump one only reaches 65. Regarding runtimes, Bayesnet is found to be four times slower than the Hoeffding Tree which is itself three times slower than Naïve Bayes and Decision Stump.

The work in [87] compares ensemble classifiers on imbalanced data streams with concept drifts, using two real datasets (Electrical, Intrusion), synthetic datasets, and six classifiers,

including the Naïve Bayes and the Hoeffding Tree ones. The Hoeffding Tree is found to be the second most accurate classifier after the Accuracy Updated Ensemble.

The authors in [41] have analyzed the resource trade-offs of six online decision trees applied to edge computing. Their results showed that the Very Fast Decision Tree and the Strict Very Fast Decision Tree were the most energy-friendly, the latter having the smallest memory footprint. On the other hand, the best predictive performances were obtained in combination with OLBoost. In particular, the paper reports an accuracy of 89.6% on the Electrical dataset, and 83.2% on an Hyperplane dataset.

Finally, the work in [21] describes the architecture of StreamDM-C++ and presents an extensive benchmark of tree-based classifiers, covering runtime, memory, and accuracy. Compared to other tree-based classifiers, the Hoeffding Tree classifier is found to have the smallest memory footprint while the Hoeffding Adaptive Tree classifier is found to be the most accurate on most of the datasets.

4.2.2 Offline and data stream classifiers for Human Activity Recognition

In this study we focus on Human Activity Recognition conducted from wearable sensors used for instance during sport activities. Other works focus on daily human activities such as cooking or cleaning [106, 7, 28], mostly using home sensors. These studies describe how they have collected datasets from heterogeneous sensor networks located in appartments. Except for the Opportunity datasets shown in [28], the data were collected from sensor placed in various daily objects such as the entrance door or the cupboard. The Opportunity dataset have, in addition, wearable sensors placed on the subjects. These papers also provide a baseline F1-score with popular classifiers (Naïve Bayes, k-NN, and neural network).

Several other studies evaluated classifiers for Human Activity Recognition with sport activities in an offline (non data stream) setting. In particular, the work in [57] compared 293 classifiers using various sensor placements and window sizes, concluding on the superiority of k nearest neighbors (k-NN) and pointing out a trade-off between runtime and classification performance. Resource consumption, including memory and runtime, was also studied for offline classifiers, such as in [68] for the particular case of the R programming language.

In addition, the work in [105] achieved an offline accuracy of 99.4% on a five-class dataset of Human Activity Recognition. The authors used AdaBoost, an ensemble method, with ten offline decision trees. The work in [3] proposes a Support Vector Machine enhanced with

feature selection. Using smartphone data, the model showed above 90% accuracy on day-to-day human activities. Finally, the work in [94] applies three offline classifiers to smartphone and smartwatch human activity data. Results show that Convolutional Neural Network and Random Forest achieve F1 score of 0.98 with smartwatches and 0.99 with smartphones.

In a data stream (online) setting, the work in [95] presents a wearable system capable of running pre-trained classifiers on the chip with high classification accuracy. It shows the superiority of the proposed Feedforward Neural Network over k-NN.

In this study, we focus on Human Activity Recognition with wearable sensor data processed with data streams classifiers and we use OrpailleCC, an OS-independent library that could be deployed on connected objects. To our knowledge, this is the first benchmark conducted in this context.

4.3 Materials and Methods

We evaluate 5 classifiers implemented in either StreamDM-C++ [21] or OrpailleCC [63]. StreamDM-C++ is a C++ implementation of StreamDM [20], a software to mine big data streams using Apache Spark Streaming. StreamDM-C++ is usually faster than StreamDM in single-core environments, due to the overhead induced by Spark.

OrpailleCC is a collection of data stream algorithms developed for embedded devices. The key functions, such as random number generation or memory allocation, are parametrizable through class templates and can thus be customized on a given execution platform. OrpailleCC is not limited to classification algorithms, it implements other data stream algorithms such as the Cuckoo filter [40] or a multi-dimensional extension of the Lightweight Temporal Compression [73]. We extended it with a few classifiers for the purpose of this benchmark.

This benchmark includes five popular classification algorithms. The Mondrian forest (Mondrian forest) [71] builds decision trees without immediate need for labels, which is useful in situations where labels are delayed [49]. The Micro-Cluster Nearest Neighbors [103] is a compressed version of the k-nearest neighbor (k-NN) that was shown to be among the most accurate classifiers for Human Activity Recognition from wearable sensors [57]. The Naïve Bayes [72] classifier builds a table of attribute occurrence to estimate class likelihoods. The Hoeffding Tree [33] builds a decision tree using the Hoeffding Bound to estimate when the best split is found. Finally, Neural Network classifiers have become popular by reaching or even exceeding human performance in many fields such as image recognition or game playing. We use a Feedforward Neural Network (Feedforward Neural Network) with one hidden

layer, as described in [95] for the recognition of fitness activities on a low-power platform.

The remainder of this section details the datasets, classifiers, evaluation metrics and parameters used in our benchmark.

4.3.1 Datasets

To conduct our benchmark, we selected the main two datasets commonly used to evaluate Human Activity Recognition from wearable sensors. In addition, we used the popular MOA stream simulator to generate three synthetic datasets with different properties.

The Banos *et al* dataset [9] is a human activity dataset with 17 participants and 9 sensors per participant[1]. Each sensor samples a 3D acceleration, gyroscope, and magnetic field, as well as the orientation in a quaternion format, producing a total of 13 values. Sensors are sampled at 50 Hz, and each sample is associated with one of 33 activities. In addition to the 33 activities, an extra activity labeled 0 indicates no specific activity.

We pre-process the Banos *et al* dataset using non-overlapping windows of one second (50 samples), and using only the 6 axes (acceleration and gyroscope) of the right forearm sensor. We compute the average and the standard deviation over the window as features for each axis,similar to the second feature set in [13]. Indeed, this feature set was providing the best performance while minimizing the producing cost.. We assign the most frequent label to the window. The resulting data points were shuffled uniformly.

In addition, we construct another dataset from Banos *et al*, in which we simulate a concept drift by shifting the activity labels in the second half of the data stream. This is useful to observe any behavioral change induced by the concept drift such as an increase in power consumption.

The Recofit dataset [79] is a human activity dataset containing 94 participants[2]. Similarly to the Banos *et al* dataset, the activity labeled 0 indicates no specific activity. Since many of these activities were similar, we merged some of them together using the same logic as in Table 4.1.

We pre-processed the dataset similarly to the Banos *et al* one, using non-overlapping windows of one second, and only using 6 axes (acceleration and gyroscope) from one sensor. . From these 6 axes, we used the average and the standard deviation over the window as features. We assigned the most frequent label to the window.

[1] Banos *et al* dataset available here.

[2] Recofit dataset available here.

Activity	Label	Activity	Label	Activity	Label
Arm band adjustment	0 (Noise)	Lawnmower (both)	19	Squat (arms in front)	32
Arm straight up	0 (Noise)	Lawnmower (left)	0 (Noise)	Squat (hands behind head)	32
Band Pull-Down row	1	Lawnmower (right)	20	Squat (kettlebell)	32
Bicep Curl	2	Lunge (both legs)	21	Squat Jump	32
Bicep Curl (band)	2	Ball Slam	22	Squat Rack Shoulder Press	32
Box Jump	3	No Exercise	0 (Noise)	Static Stretching	0 (Noise)
Burpee	4	Note	0 (Noise)	Stretching	0 (Noise)
Butterfly sit-up	5	Triceps Extension (standing)	23	Tap IMU	0 (Noise)
Chest Press	6	Triceps Extension (both)	23	Tap left IMU	0 (Noise)
Crunch	7	Plank	24	Tap right IMU	0 (Noise)
Device on Table	0 (Noise)	Power Boat pose	25	Triceps Kickback (bench–both)	33
Dip	8	Pushups (foot variation)	26	Triceps Kickback (bench–left)	0 (Noise)
Dumbbell Deadlift Row	9	Pushups	26	Triceps Kickback (bench–right)	33
Dumbbell Row (both)	10	Stretching	0 (Noise)	Triceps Extension (lying–both)	34
Dumbbell Row (left)	0 (Noise)	Rest	0 (Noise)	Triceps Extension (lying–left)	0 (Noise)
Dumbbell Row (right)	11	Rowing Machine	27	Triceps Extension (lying–right)	34
Dumbbell Squat (hands at side)	12	Running	28	Two-arm Dumbbell Curl (both)	35
Dynamic Stretch	0 (Noise)	Russian Twist	29	Non-listed	0 (Noise)
Elliptical Machine	13	Seated Back Fly	30	V-up	36
Punches	14	Shoulder Press	31	Walk	37
Invalid	0 (Noise)	Side Plank (left)	24	Walking lunge	38
Jump Rope	15	Side Plank (right)	24	Wall Ball	39
Jumping Jacks	16	Sit-up (hand behind head)	5	Wall Squat	40
Kettlebell Swing	17	Sit-up	5	Dumbbell Curl (alternating)	35
Lateral Raise	18	Squat	32		

Table 4.1: Activity merging in Recofit dataset (table extracted from [31]).

Massive Online Analysis [18] (MOA) is a Java framework to compare data stream classifiers. In addition to classification algorithms, MOA provides many tools to read and generate datasets. We generate three synthetic datasets[3]: a hyperplane, a RandomRBF, and a RandomTree dataset. We generate 200,000 data points for each of these synthetic datasets. The hyperplane and the RandomRBF both have three features and two classes, however, the RandomRBF has a slight imbalance toward one class. The RandomTree dataset is the hardest of the three, with six attributes and ten classes. Since the data points are generated with a tree structure, we expect the decision trees to show better performances than the other classifiers.

4.3.2 Algorithms and Implementation

In this section, we describe the algorithms used in the benchmark, their hyperparameters, and relevant implementation details. We selected two of the main algorithms in the popular StreamDM library: Naïve Bayes and Hoeffding Tree. These algorithms are commonly found in data stream classification studies. In addition, we selected representative algorithms from the main classfification approaches: the Mondrian forest for tree based-learning, Micro Cluster Nearest Neighbor for cluster based-learning, and a Feedforward Neural Network for neural networks.

Mondrian forest (Mondrian forest) [71]

Each tree in a Mondrian forest recursively splits the feature space, similar to a regular decision tree. However, the feature used in the split and the value of the split are picked randomly. The probability to select a feature is proportional to its normalized range, and the value for the split is uniformly selected in the range of the feature. During prediction, a node combines its observed label count with its parent prediction. Since the Mondrian tree is able to reshape the internal structure of the tree, we expect the Mondrian forest to recover from concept drifts, although most likely slower than MCNN.

In OrpailleCC, the amount of memory allocated to the forest is predefined, and it is shared by all the trees in the forest, leading to a constant memory footprint for the classifier. This implementation is memory-bounded, meaning that the classifier can adjust to memory limitations, for instance by stopping tree growth or replacing existing nodes with new ones.

[3] MOA commands available here.

Number of trees	Base count	Discount	Budget
1	0.0	1.0	1.0
5	0.0	1.0	0.4
10	0.0	1.0	0.4
50	0.0	1.0	0.2

Table 4.2: Hyperparameters used for the Mondrian forest.

This is different from an implementation with a constant space complexity, where the classifier would use the same amount of memory regardless of the amount of available memory. For instance, in our study, the Mondrian forest classifier is memory-bounded while Naïve Bayes classifier has a constant space complexity.

Mondrian trees can be tuned using three parameters: the base count, the discount factor, and the budget. The base count is used to initialize the prediction for the root. The discount factor influences the nodes on how much they should use their parent prediction. A discount factor closer to one makes the prediction of a node closer to the prediction of its parent. Finally, the budget controls the tree depth.

Hyperparameters used for Mondrian forest are shown in Table 4.2. Additionally, the Mondrian forest is allocated with 600 KB of memory unless specified otherwise. On the Banos *et al* and Recofit datasets, we also explore the Mondrian forest with 3 MB of memory in order to observe the effect of available memory on performances (classification, runtime, and power).

Micro Cluster Nearest Neighbor [103]

The Micro Cluster Nearest Neighbor (MCNN) is a variant of k-nearest neighbors where data points are aggregated into clusters to reduce storage requirements. During training, the algorithm merges a new data point to the closest cluster that shares the same label. If the closest cluster does not share the same label as the data point, this closest cluster and the closest cluster with the same label are assigned an error. When a cluster receives too many errors, it is split. During classification, MCNN returns the label of the closest cluster. Regularly, the algorithm also assigns a participation score to each cluster and when this score gets below a threshold, the cluster is removed. Given that the maximum number of clusters is fixed, this mechanism makes space for new clusters, and possibly helps adjust to concept drifts. The space and time complexities of MCNN are constant since the maximum number

Number of clusters	Error threshold	Participation threshold
10	2	10
20	10	10
33	16	10
40	8	10
50	2	10

Table 4.3: Hyperparameters used for the MCNN.

of clusters is fixed. The reaction to concept drift is influenced by the participation threshold and the error threshold. A higher participation threshold and a lower error threshold increase reaction speed to concept drift. Since the error thresholds used in this study are small, we expect MCNN to react quite fast and efficiently to concept drifts.

We implemented two versions of MCNN in OrpailleCC, differing in the way they remove clusters during training. The first version (MCNN Origin) is similar to the mechanism described in [103], based on participation scores. The second version (MCNN OrpailleCC) removes the cluster with the lowest participation only when space is needed. A cluster slot is needed when an existing cluster is split and there is no more slot available because the number of active clusters already reached the maximum defined by the user.

MCNN OrpailleCC has only one parameter, the error threshold after which a cluster is split. MCNN Origin has two parameters: the error threshold and the participation threshold. The participation threshold is the limit below which a cluster is removed. Hyperparameters used for MCNN are shown in Table 4.3. Additionally, they both implementations have the cluster count as their last parameter.

Naïve Bayes (Naïve Bayes) [72]

The Naïve Bayes algorithm keeps a table of counters for each feature value and each label. During prediction, the algorithm assigns a score for each label depending on how the data point to predict compares to the values observed during the training phase. Since the counters are not biased toward the more recent data points, we expect Naïve Bayes to be slow to adapt if not ineffective in a concept drift situation.

The implementation from StreamDM-C++ was used in this benchmark. It uses a Gaussian fit for numerical attributes. Two implementations were used, the OrpailleCC one and the StreamDM one. We used two implementations to provide a comparison reference between

the two libraries.

Hoeffding Tree (Hoeffding Tree) [33]

Similar to a decision tree, the Hoeffding Tree recursively splits the feature space to maximize a metric, often the information gain or the Gini index. However, to estimate when a leaf should be split, the Hoeffding Tree relies on the Hoeffding bound, a measure of the score deviation of the splits. This measure allows the leaf to decide when the best split is clearly better than the second best split based on the data observed from the stream. This mechanism prevents the tree from waiting until the end of the stream to ensure a split is the best. During classification, a data point is sorted to a leaf, and a label is predicted by aggregating the labels of the training data points in that leaf, usually through majority voting or Naïve Bayes classification. We used this classifier as implemented in StreamDM-C++. The Hoeffding Tree is common in data stream classification, however, the internal nodes are static and cannot be re-considered. Therefore, any concept drift adaption relies on the new leaves that will be split.

The Hoeffding Tree has three parameters: the confidence level, the grace period, and the leaf learner. The confidence level is the probability that the Hoeffding bound makes a wrong estimation of the deviation. The grace period is the number of processed data points before a leaf is evaluated for a split. The leaf learner is the method used in the leaf to predict the label. In this study, we used a confidence level of 0.01 with a grace period of 10 data points and the Naïve Bayes classifier as leaf learner.

Feedforward Neural Network (Feedforward Neural Network)

A neural network is a combination of artificial neurons, also known as perceptrons, that all have input weights and an activation function. To predict a class label, the perceptron applies the activation function to the weighted sum of its input values. The output value of the perceptron is the result of this activation function. This prediction phase is also called feed-forward. To train the neural network, feed-forward is applied first, then the error between the prediction and the expected result is used in the backpropagation process to adjust the weights of the input values. A neural network combines multiple perceptrons by connecting perceptron outputs to inputs of other perceptrons. One of the key benefits of the neural network is that it maintains a consistent memory footprint, making it ideal for processing data streams. In this benchmark, we used a fully-connected Feedforward Neural Network, that is, a network where perceptrons are organized in layers and all output values

from perceptrons of layer $n-1$ serve as input values for perceptrons of layer n. We used a 3-layer network with 120 inputs, 30 perceptrons in the hidden layer, and 33 output perceptrons. Because a Feedforward Neural Network takes many epochs to update and converge it barely adapts to a concept drifts even though it trains with each new data point. Instead of the features described in Section 4.3.1, we utilized histogram features from [95] in this study as the network did not perform well with the former. These histogram features generate 20 bins per axis.

This neural network can be tuned by changing the number of layers and the size of each layer. Additionally, the activation function and the learning ratio can be changed. The learning ratio indicates by how much the weights should change during backpropagation.

Hyperparameters Tuning

For each classifier, we tuned hyperparameters using the first subject from the Banos *et al* dataset. The data from this subject was pre-processed as the rest of the Banos *et al* dataset (window size of one second, average and standard deviation on the six-axis of the right forearm sensor, ...). We did a grid search to test multiple values for the parameters.

The classifiers start the prequential phase with no knowledge from the first subject. We made an exception for the Feedforward Neural Network because we noticed that it performed poorly with random weights and it needed many epochs to achieve better performances than a random classifier. Therefore, we decided to pre-train the Feedforward Neural Network and re-use the weights as a starting point for the prequential phase.

For other classifiers, only the hyperparameters were taken from the tuning phase. We selected the hyperparameters that maximized the F1 score on the first subject.

Offline Comparison

We compared data stream algorithms with an offline k-NN. The value of k were selected using a grid search. The k-NN F1 score computation is computed offline using a test set comprising 10% of the stream, while the remaining 90% is used as the training set.

4.3.3 Evaluation

We computed four metrics: the F1 score, the memory footprint, the runtime, and the power usage. The F1 score and the memory footprint were computed periodically during the execution of a classifier. The power consumption and the runtime were collected at the

end of each execution.

To compute the F1 score, we monitor the true positives, false positives, true negatives, and false negatives using the prequential evaluation, meaning that with each new data point the model is first tested and then trained. From these counts, we compute the F1 score every 50 elements. We do not apply any fading factor to attenuate errors throughout the stream. We compute the F1 score in a one-versus-all fashion for each class, averaged across all classes (macro-average, code available here). When a class has not been encountered yet, its F1 score is ignored. We use the F1 score rather than the accuracy because the real data sets are imbalanced.

We measure the memory footprint by reading file `/proc/self/statm` every 50 data points.

The runtime of a classifier is the time needed for the classifier to process the dataset. We collect the runtime reported by the *perf* command[4], which includes loading of the binary in memory, setting up data structures, and opening the dataset file. To remove these overheads from our measurements, we use the runtime of an empty classifier that always predict class 0 as a baseline.

We measure the average power consumed by classification algorithms with the *perf* command. The power measurement is done multiple times in a minimal environment. We use the empty classifier as a baseline.

4.3.4 Results Reproducibility

The code and datasets used in this study are available on Github at `https://github.com/big-data-lab-team/benchmark-har-data-stream.git`. We used release 1.0, available on Zenodo with DOI 10.5281/zenodo.4148947 for long-term archival. The repository contains the source code and data we used to run the experiment, including the Makefile to compile the benchmark, the Python script to organize the execution and plot the results, and the datasets. This experiment requires the following software to run properly.

- `Git` to download the repository and its submodules.
- `gcc` or any C++ compiler in combination with `make` to compile the benchmark binaries.
- `log4cpp` as a dependency of StreamDM-C++.
- `Python`, `seaborn`, `matplotlib`, and `pandas`, to run the experiment and plot the results.

[4] *perf* website

The plotting script (`makefile.py`) extracts data from three CSV files: `models.csv`, `output_runs`, and `output`. `models.csv` lists the combination of classifiers, datasets, and parameters that were run. This combination is called a model. The `output_runs` file stores information about the model's repetition such as the runtime or the energy. Finally, the `output` file contains the accuracy, the F1 score, and the memory footprint every fifty elements. Each line is identified with three ids: the model id, the repetition count, and the data point count in the dataset.

In the README file, we provided the commands to download, compile, run the experiment, and plot the results.

During the benchmark execution, the datasets and output files were stored in memory through a memfs filesystem mounted on `/tmp`, to reduce the impact of I/O time. We averaged metrics across repetitions (same classifier, same parameters, and same dataset). The experiment was done with a single core of a cluster node with two Intel(R) Xeon(R) Gold 6130 CPUs and a main memory of 250G. The node was running a CentOS Linux 8 with Linux kernel 4.18.

4.4 Results

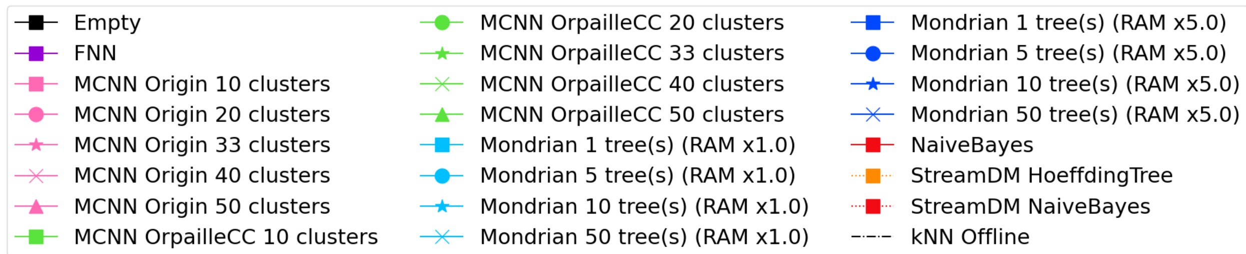
This section presents our benchmark results and the corresponding hyperparameter tunning experiments.

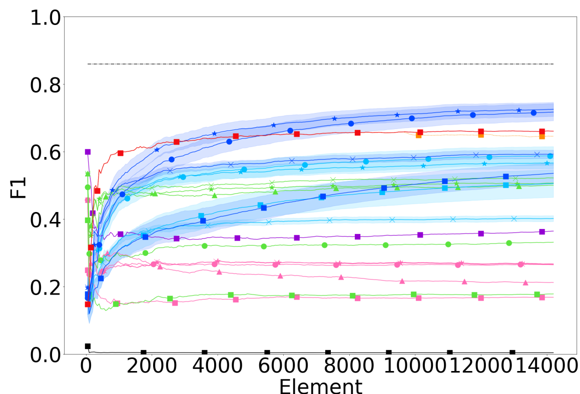
4.4.1 Overall classification performance

Figure 4.1 compares the F1 scores obtained by all classifiers on the six datasets. The graphs also show the standard deviation of the Mondrian forest classifier observed across all repetitions (the other classifiers do not involve any randomness). Table 4.4 also shows the last F1 scores obtained on each dataset.

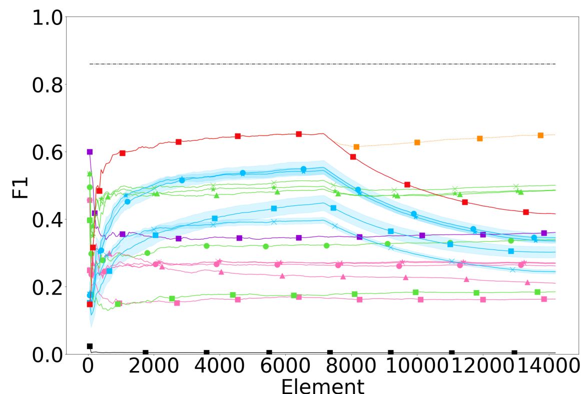
F1 scores vary greatly across the datasets. While the highest observed F1 score is above 0.95 on the Hyperplane and RandomRBF datasets, it barely reaches 0.65 for the Banos *et al* dataset, and it remains under 0.40 on the Recofit and RandomTree datasets. This trend is consistent for all classifiers.

The offline k-NN classifier used as baseline achieves better F1 scores than all other classifiers, except for the Mondrian forest on the Hyperplane and the RandomRBF datasets. On the Banos *et al* dataset, the difference of 0.23 with the best stream classifier remains

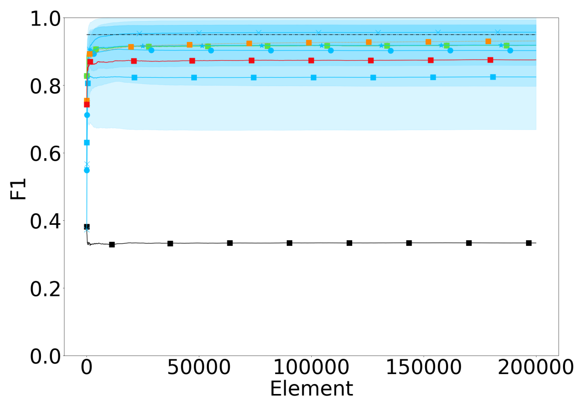




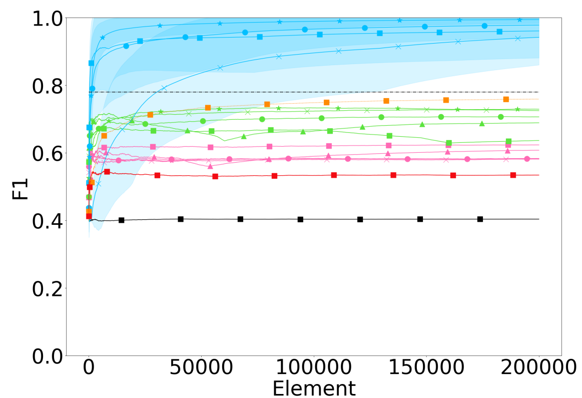
(a) Banos *et al*



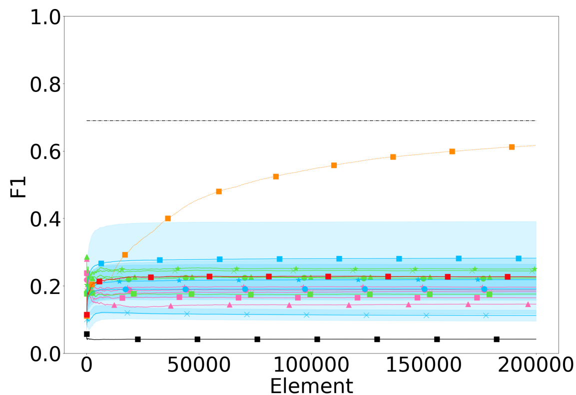
(b) Banos *et al* (with Drift)



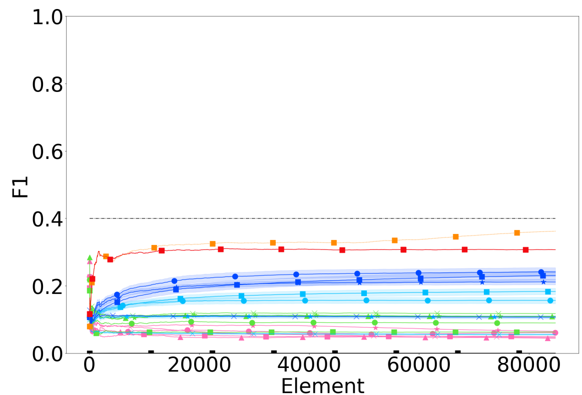
(c) Hyperplane (MOA)



(d) RandomRBF (MOA)



(e) RandomTree (MOA)



(f) Recofit

Figure 4.1: F1 scores for the six datasets (average over 20 rep.). The horizontal dashed black line indicates the offline k-NN F1 score (the value of k was obtained by grid search in [2, 20]). The blue shades is the $\pm\sigma$ interval of the Mondrian forest classifier.

	Hyperplane	RandomRBF	RandomTree	Recofit	Banos *et al*	Banos *et al* (drift)
Empty	0.333	0.404	0.041	0.000	0.004	0.004
MCNN Origin 10 clusters	0.918	0.624	0.164	0.050	0.169	0.163
MCNN Origin 20 clusters	0.918	0.583	0.188	0.061	0.265	0.262
MCNN Origin 33 clusters	0.918	0.584	0.196	0.065	0.268	0.270
MCNN Origin 40 clusters	0.918	0.581	0.183	0.047	0.266	0.268
MCNN Origin 50 clusters	0.918	0.607	0.146	0.045	0.212	0.210
MCNN OrpailleCC 10 clusters	0.918	0.637	0.176	0.063	0.178	0.185
MCNN OrpailleCC 20 clusters	0.918	0.707	0.222	0.090	0.332	0.339
MCNN OrpailleCC 33 clusters	0.918	0.729	0.250	0.109	0.507	0.487
MCNN OrpailleCC 40 clusters	0.918	0.725	0.244	0.118	0.522	0.501
MCNN OrpailleCC 50 clusters	0.918	0.689	0.226	0.109	0.500	0.485
Mondrian 1 tree(s) (RAM x1.0)	0.825	0.961	0.282	0.183	0.506	0.302
Mondrian 5 tree(s) (RAM x1.0)	0.903	0.978	0.190	0.157	0.588	0.345
Mondrian 10 tree(s) (RAM x1.0)	0.919	0.994	0.218	0.107	0.566	0.336
Mondrian 50 tree(s) (RAM x1.0)	0.957	0.942	0.112	0.056	0.402	0.244
NaiveBayes	0.875	0.534	0.227	0.307	0.661	0.416
StreamDM HoeffdingTree	0.931	0.759	0.617	0.362	0.656	0.650
StreamDM NaiveBayes	0.875	0.534	0.227	0.311	0.661	0.416
Mondrian 1 tree(s) (RAM x5.0)				0.230	0.536	
Mondrian 5 tree(s) (RAM x5.0)				0.241	0.717	
Mondrian 10 tree(s) (RAM x5.0)				0.212	0.726	
Mondrian 50 tree(s) (RAM x5.0)				0.107	0.593	
Feedforward Neural Network					0.365	0.360

Table 4.4: Average F1 scores obtained on the last data point of the stream.

very substantial, which quantifies the remaining performance gap between data stream and offline classifiers. On the Recofit dataset, the difference between stream and offline classifiers is lower, but the offline performance remains very low.

It should be noted that the F1 scores observed for the offline k-NN classifier on the real datasets are substantially lower than the values reported in the literature. On the Banos *et al* dataset, the original study in [13] reports an F1 score of 0.96, the work in [31] achieves 0.92, but our benchmark only achieves 0.86. Similarly, on the Recofit dataset, the original study reports an accuracy of 0.99 and the work in [31] reaches 0.65 while our benchmark only achieves 0.40. This is most likely due to our use of data coming from a single sensor, consistently with our motivating use case, while the other works used multiple ones (9 in the case of Banos *et al*).

The Hoeffding Tree appears to be the most robust to concept drifts (Banos *et al* with drift), while the Mondrian forest and Naïve Bayes classifiers are the most impacted. MCNN classifiers are marginally impacted. The low resilience of Mondrian forest to concept drifts can be attributed to two factors. First, existing nodes in trees of a Mondrian forest cannot be updated. Second, when the memory limit is reached, Mondrian trees cannot grow or reshape their structure anymore.

4.4.2 Hoeffding Tree and Naïve Bayes

The Naïve Bayes and the Hoeffding Tree classifiers stand out on the two real datasets (Banos *et al* and Recofit) even though the F1 scores observed remain low (0.6 and 0.35) compared to offline k-NN (0.86 and 0.40). Additionally, the Hoeffding Tree performs outstandingly on the RandomTree dataset and Banos *et al* dataset with a drift. Such good performances were expected on the RandomTree dataset because it was generated based on a tree structure.

Except for the Banos *et al* dataset, the Hoeffding Tree performs better than Naïve Bayes. For all datasets, the performance of both classifiers is comparable at the beginning of the stream, because the Hoeffding Tree uses a Naïve Bayes in its leaves. However, F1 scores diverge throughout the stream, most likely because of the Hoeffding Tree's ability to reshape its tree structure. This occurs after a sufficient amount of elements, and the difference is more noticeable after a concept drift.

Finally, we note that the StreamDM-C++ and OrpailleCC implementations of Naïve Bayes are indistinguishable from each other, which confirms the correctness of our implementation in OrpailleCC.

4.4.3 Mondrian forest

On two synthetic datasets, Hyperplane and RandomRBF, the Mondrian forest (RAM x 1.0) with 10 trees achieves the best performance (F1>0.95), above offline k-NN. Additionally, the Mondrian forest with 5 or 10 trees ranks third on the two real datasets.

Surprisingly, a Mondrian forest with 50 trees performs worse than 5 or 10 trees on most datasets. The only exception is the Hyperplane dataset where 50 trees perform between 5 and 10 trees. This is due to the fact that our Mondrian forest implementation is memory-bounded, which is useful on connected objects but limits tree growth when the allocated memory is full. Because 50 trees fill the memory faster than 10 or 5 trees, the learning stops earlier, before the trees can learn enough from the data. It can also be noted that the variance of the F1 score decreases with the number of trees, as expected.

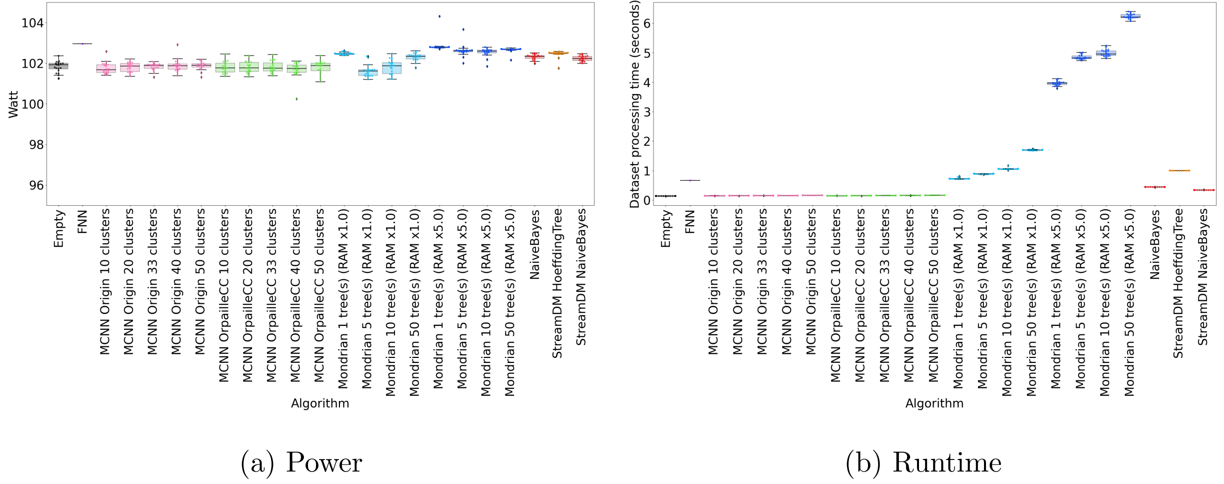
The dependency of the Mondrian forest to memory allocation is shown in Banos *et al* and Recofit datasets, where an additional configuration with five times more memory than the initial configuration was run (total of 3 MB). The memory increase induces an F1 score difference greater than 0.1, except when only one tree is used, in which case the improvement caused by the memory is less than 0.05. Naturally, the selected memory bound may not be achievable on a connected object. Overall, Mondrian forest seems to be a viable alternative to Naïve Bayes or the Hoeffding Tree for Human Activity Recognition.

4.4.4 MCNN

The MCNN OrpailleCC stands out on the Banos *et al* (with drift) dataset where it ranks second thanks to its adaptation to the concept drift. On other datasets, MCNN OrpailleCC ranks below the Mondrian forest and the Hoeffding Tree, but above MCNN Original. This difference between the two MCNN implementations is presumably due to the fact that MCNN Origin removes clusters with low participation too early. On the real datasets (Banos *et al* and Recofit), we notice that the MCNN OrpailleCC appears to be learning faster than the Mondrian forest, although the Mondrian forest catches up after a few thousand elements. Finally, we note that MCNN remains quite lower than the offline k-NN.

4.4.5 Feedforward Neural Network

Figure 4.1a shows that the Feedforward Neural Network has a low F1 score (0.36) compared to other classifiers (above 0.5), which contradicts the results reported in [95] where



(a) Power (b) Runtime

Figure 4.2: Power usage and Runtime (20 repetitions) for the Banos *et al* dataset. Results are similar across datasets.

the Feedforward Neural Network achieves more than 95% accuracy in a context of offline training. The main difference between [95] and our study lies in the definition of the training set. In [95], the training set includes examples from every subject, while we only use a single one, to ensure an objective comparison with the other stream classifiers that do not require offline training (except for hyperparameter tuning, done on the first subject of the Banos *et al* dataset). When we use a random sample of 10% of the datapoints across all subjects for offline training, we reach an F1 score of 0.68, which is higher than the performance of the Naïve Bayes classifier.

4.4.6 Power

Figure 4.2a shows the power usage of each classifier on four datasets (results are similar for the other two datasets). All classifiers exhibit comparable power consumptions, close to 102 W.

This observation is explained by two factors. First, the benchmarking platform was working at minimal power. To ensure no disturbance by a background process, we run the classifiers on an isolated cluster node with eight cores. Therefore, the power difference on one core is not noticeable.

Another reason is the dataset sizes. Indeed, the slowest run is about 10 seconds with 50 Mondrian trees on Recofit dataset. Such short executions do not leave time for the CPU to switch P-states because it barely warms a core. Further experiments would be required to

check how our power consumption observations generalize to connected objects.

4.4.7 Runtime

Figure 4.2b shows the classifier runtimes for the two real datasets. The Mondrian forest is the slowest classifier, in particular for 50 trees which reaches 2 seconds on Banos *et al* dataset. This represents roughly 0.35 ms/element with a slower CPU. The second slowest classifier is the Hoeffding Tree, with a runtime comparable to the Mondrian forest with 10 trees. The Hoeffding Tree is followed by the two Naïve Bayes implementations, which is not surprising since Naïve Bayes classifiers are used in the leaves of the Hoeffding Tree. The MCNN classifiers are the fastest ones, with a runtime very close to the empty classifier. Note that allocating more memory to the Mondrian forest substantially increases runtime.

We observe that the runtime of StreamDM-C++'s Naïve Bayes is comparable to OrpailleCC's. This suggests that the performance of the two libraries is similar, which justifies our comparison of Hoeffding Tree and Mondrian forest.

4.4.8 Memory

Figure 4.3 shows the evolution of the memory footprint for the Banos *et al* dataset. Results are similar for the other datasets and are not reported for brevity. Since the memory footprint of the Naïve Bayes classifier was almost indistinguishable from the empty classifier, we used the two Naïve Bayes as a baseline for the two libraries. This enables us to remove the 1.2 MB overhead induced by StreamDM-C++. The StreamDM-C++ memory footprint matches the result in [21], where the Hoeffding Tree shows a memory footprint of 4.8 MB.

We observe that the memory footprints of the Mondrian forest and the Hoeffding Tree are substantially higher than for the other classifiers, which makes their deployment on connected objects challenging. Overall, memory footprints are similar across datasets, due to the fact that most algorithms follow a bounded memory policy or have a constant space complexity. The only exception is the Hoeffding Tree that constantly selects new splits depending on new data points. The Mondrian forest has the same behavior but the OrpailleCC implementation is memory-bounded, which makes its memory footprint constant.

4.4.9 Hyperparameter tuning

Figure 4.4 shows the impact of the error threshold in the MCNN classifiers with different cluster counts. The error threshold of MCNN has little impact on classification performance.

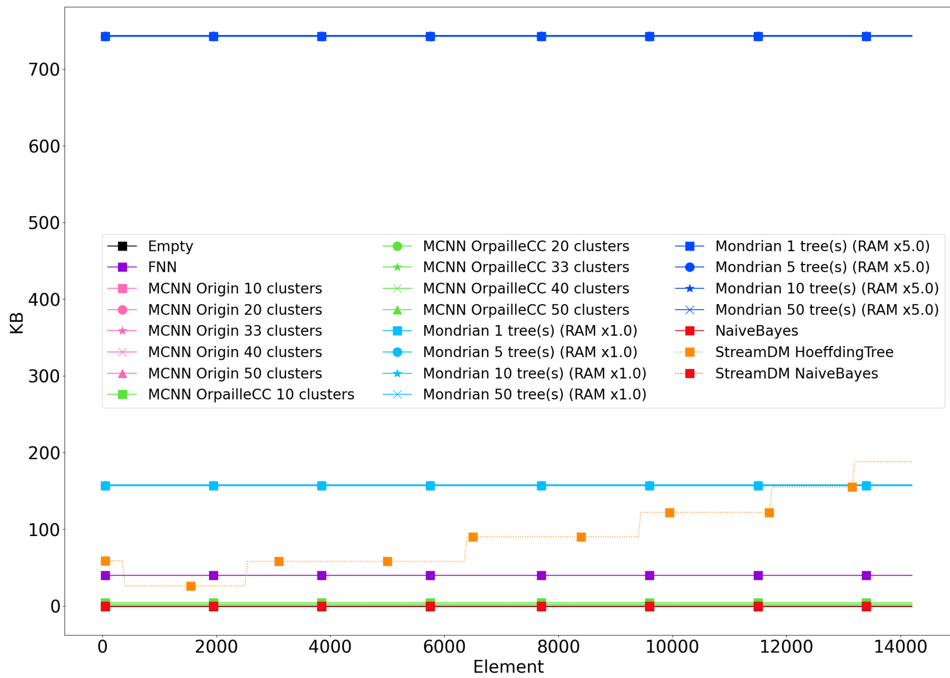


Figure 4.3: Memory footprint of classifiers with the empty classifier as a baseline, measured on the Banos *et al* dataset. The memory footprint of the empty classifier is 3.44 MB. The baselines are the two Naïve Bayes from OrpailleCC and StreamDM-C++. Their respective memory footprints are 3.44 MB and 4.74 MB.

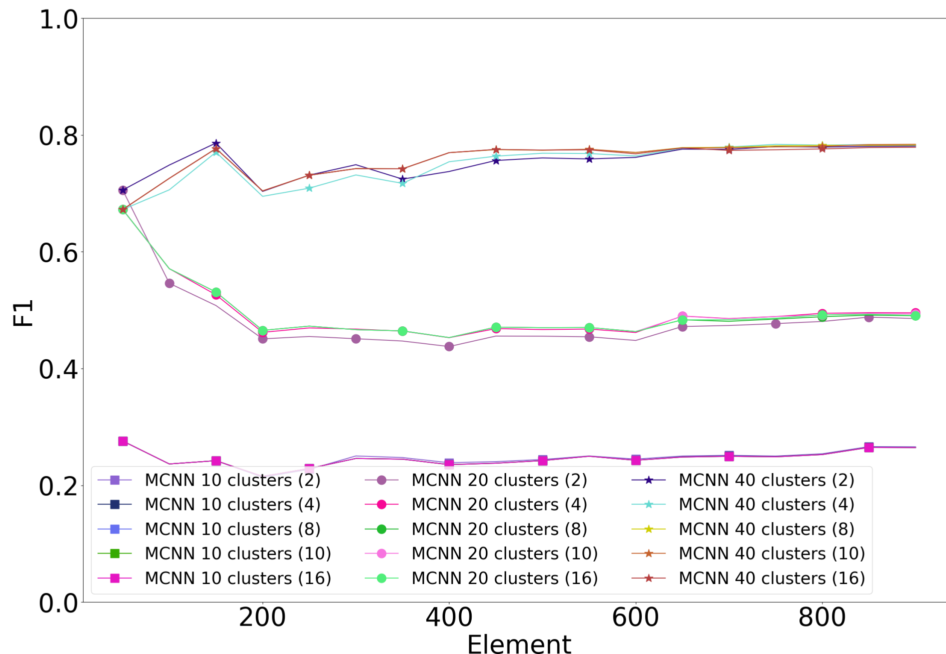


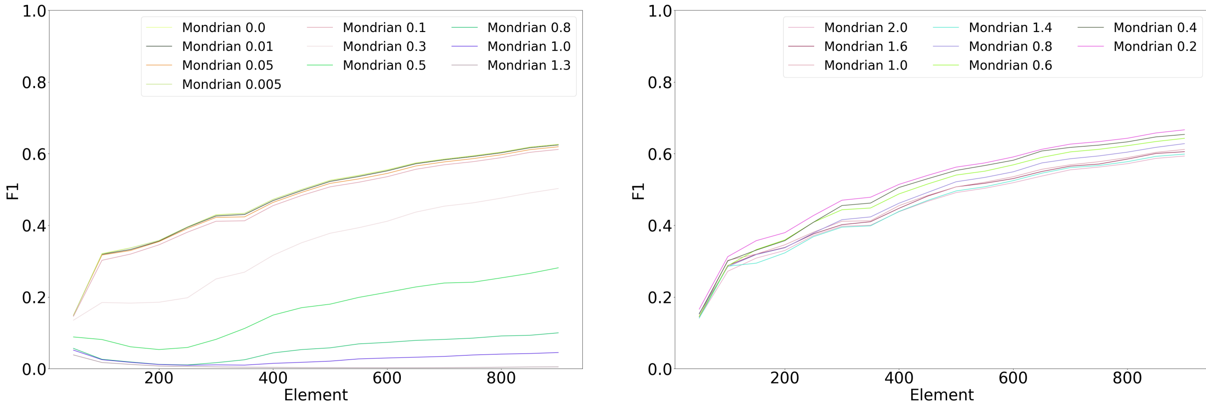
Figure 4.4: Error threshold tuning of MCNN with the first subject of Banos *et al* dataset. Error threshold in parenthesis.

For 20 and 40 clusters, the best-performing threshold is either 2 or 4, meaning that a cluster may do 2 or 4 errors before being split. For 10 clusters, all error thresholds perform equally.

Figure 4.5 shows the impact of the Mondrian forest hyperparameters on the classification performance. The base count hyperparameter (Figure 4.5a) has a very substantial impact on classification performance; the smallest value (0.0) results in the best performance. On the contrary, the budget hyperparameter (Figure 4.5b) only has a moderate impact on classification; the best value is 0.2. Finally, the discount hyperparameter (Figure 4.5c) has a negligible impact on the performance; the best-performing value is 0.1.

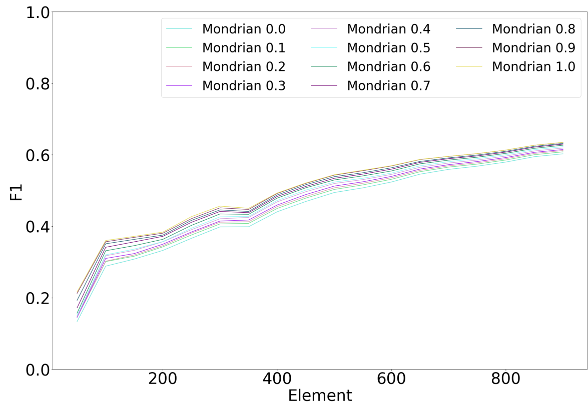
4.5 Conclusion

We conclude that the Hoeffding Tree, the Mondrian forest, and the Naïve Bayes data stream classifiers have an overall superiority over the Feedforward Neural Network and the MCNNs ones for Human Activity Recognition. However, the prediction performance remains quite low compared to an offline k-NN classifier, and it varies substantially between datasets. Noticeably, the Hoeffding Tree and the MCNNs classifiers are more resilient to concept drift that the other ones.



(a) Impact of the base count with 10 trees, a budget of 1.0, and a discount factor of 0.2.

(b) Impact of the budget with 10 trees, a base count of 0.1, and discount factor of 0.2.



(c) Impact of the discount factor with 10 trees, a budget of 1.0, and a base count of 0.1.

Figure 4.5: Hyperparameters tuning for Mondrian with the first subject of Banos *et al* dataset.

Regarding memory consumption, only the MCNN and Naïve Bayes classifiers were found to have a negligible memory footprint, in the order of a few kilobytes, which is compatible with connected objects. Conversely, the memory consumed by a Mondrian forest, a Feed-forward Neural Network or a Hoeffding Tree is in the order of 100 kB, which would only be available on some connected objects. In addition, the classification performance of a Mondrian forest is strongly modulated by the amount of memory allocated. With enough memory, a Mondrian forest is likely to match or exceed the performance of the Hoeffding Tree and Naïve Bayes classifiers.

The amount of energy consumed by classifiers is mostly impacted by their runtime, as all power consumptions were found comparable. The Hoeffding Tree and Mondrian forest are substantially slower than the other classifiers, with runtimes in the order of 0.35 ms/element, a performance not compatible with common sampling frequencies of wearable sensors.

Our results show that even though data stream classifiers were developed to minimize memory footprint, they were not developed to work with a given memory budget. For instance, there is no guarantee that the Mondrian forest or the Hoeffding Tree would make the best split selection on a device with a small amount of memory. Future research could focus on developing algorithms that can guarantee a peak memory consumption and ensure an optimal response within a memory budget. In particular, we are planning to explore tree sampling methods based on memory and performance criteria in Mondrian forest. In addition, deploying classification algorithms on actual connected objects might highlight other relevant research directions.

Chapter 5

Mondrian Forest for Data Stream Classification Under Memory Constraints

Published as: Martin Khannouz and Tristan Glatard. Mondrian Forest for Data Stream Classification Under Memory Constraints. *arXiv:2205.07871 [cs.LG]*, 2022.

This preprint is currently under revision in the *Data Mining and Knowledge Discovery* journal.

Supervised learning algorithms generally assume the availability of enough memory to store data models during the training and test phases. However, this assumption is unrealistic when data comes in the form of infinite data streams, or when learning algorithms are deployed on devices with reduced amounts of memory. In this paper, we adapt the online Mondrian forest classification algorithm to work with memory constraints on data streams. In particular, we design five out-of-memory strategies to update Mondrian trees with new data points when the memory limit is reached. Moreover, we design node trimming mechanisms to make Mondrian trees more robust to concept drifts under memory constraints. We evaluate our algorithms on a variety of real and simulated datasets, and we conclude with recommendations on their use in different situations: the Extend Node strategy appears as the best out-of-memory strategy in all configurations, whereas different node trimming mechanisms should be adopted depending on whether a concept drift is expected. All our methods are implemented in the OrpailleCC open-source library and are ready to be used on embedded systems and connected objects.

5.1 Introduction

Supervised classification algorithms mostly assume the availability of abundant memory to store data and models. This is an issue when processing data streams — which are infinite sequences by definition — or when using memory-limited devices as is commonly the case in the Internet of Things. This paper studies how the Mondrian forest, a popular online classification method, can be adapted to work with data streams and memory constraints compatible with connected objects.

Although online and data stream classification methods both assume that the dataset is available as a sequence of elements, data stream methods assume that the dataset is infinite whereas online methods consider that it is large but still bounded in size. Consequently, online methods usually store the processed elements for future access whereas data stream methods do not.

Under memory constraints, a data stream classification model that has reached its memory limit faces two issues: (1) how to update the model when new data points become available, which we denote as the *out-of-memory* strategy, and (2) how to adapt the model to concept drifts, i.e., changes in the learned concepts. The mechanisms described in this paper address these two issues in the Mondrian forest.

The Mondrian forest is a tree-based, ensemble, online learning method with comparable performance to offline Random Forest [71]. Previous experiments highlighted the Mondrian forest sensitivity to the amount of available memory [60], which motivates their extension to memory-constrained environments. In practice, existing implementations [12, 54] of the Mondrian forest all assume enough memory and crash when memory is not available.

The concept drift is a common problem in data streams that occurs when the distribution of features changes throughout the stream. By design, the Mondrian forest is not equipped to adapt to concept drifts as its trees cannot be pruned, trimmed, or modified. When memory is saturated, this lack of adaptability to concept drift worsens as trees cannot even grow new branches to accommodate changes in feature distributions. Therefore, the Mondrian forest needs a mechanism to free memory such that new tree nodes can grow. Such a mechanism might also be useful for stable data streams as it would replace less accurate nodes with better performing ones.

In summary, this paper makes the following contributions:

1. We adapt the Mondrian forest for data streams;

2. We propose five new out-of-memory strategies for the Mondrian forest under memory

constraints;

3. We propose three new node trimming mechanisms to make Mondrian forest adaptive to concept drifts;

4. We evaluate our strategies on six simulated and real datasets.

5.2 Materials and Methods

All the methods presented in this section are implemented in the OrpailleCC framework [63]. The scripts to reproduce our experiments are available on GitHub at https://github.com/big-data-lab-team/benchmark-har-data-stream.

5.2.1 Mondrian Forest

The Mondrian forest [71] is an ensemble method that combines Mondrian trees. Each tree in the forest recursively splits the feature space, similar to a regular decision tree. However, the feature used in the split and the value of the split are picked randomly. The probability to select a feature is proportional to its range, and the value for the split is uniformly selected in the range of the feature. In contrast with other decision trees, the Mondrian tree does not split leaves to introduce new nodes. Instead, it introduces a new parent and a sibling to the node where the split occurs. The original node and its descendant are not modified and no data point is moved to that new sibling beside the data point that initialized the split. This approach allows the Mondrian tree to introduce new branches to internal nodes. This training algorithm does not rely on labels to build the tree, however, each node maintains counters for each label seen. Therefore, labels can be delayed, but are needed before the prediction. In addition to the counters, each node keeps track of the range of its feature that represents a box that containing all data points. A data point can create a new branch only if it is sorted to a node and it falls outside of the node's box.

5.2.2 Mondrian Forest for Data Stream Classification

The implementation of Mondrian forest presented in [71, 12] is online because trees rely on potentially all the previously seen data points to grow new branches. To support data streams, the Mondrian forest has to access data points only once as the dataset is assumed to be infinite in size. Algorithm 1, adapted from reference [71], describes our data-stream

implementation. Function `update_box` (line 11) updates the node's box using the data point's features. Function `update_counters` (line 12) updates the label counters assigned to the node. Function `is_enough_memory` (line 2) returns true if there is enough memory to run `extend_tree` (line 7). Function `distance_to_box` (line 3) compute the distance between a data point and the node's box and it returns zero if the data point fall inside the box. Function `random_bool` (line 5) randomly select a boolean that is more likely to be true when the data point is far from the node's box. This function always return false if the distance is zero. Finally, function `extend_tree` extends the tree by introducing a new parent and a new sibling to the current node.

To make a prediction, a node assigns a score to each label. This score uses the node's counters, the parent's scores, and the score generated by an hypothetical split. The successive scores of the nodes encountered by the test data point are combined together to create the tree score. Finally, the scores of the trees are averaged to get the forest's score. The prediction is the label with the highest score.

Mondrian trees can be tuned using three parameters: the base count, the discount factor, and the budget. The base count is a default score given to the root's parent. The discount factor controls the contribution of a node to the score of its children. A discount factor closer to one makes the prediction of a node closer to the prediction of its parent. Finally, the budget controls the tree depth. A small budget makes nodes virtually closer to each other, and thus, less likely to introduce new splits.

5.2.3 Out-of-memory Strategies in the Mondrian Tree

A memory-constrained Mondrian forest needs to determine what to do with data points when the memory limit is reached. We designed five out-of-memory strategies for this purpose. These strategies specify how the statistics, namely the counters and the box limits, should be updated when training nodes. They are implemented in functions `update_box` and `update_counters` called at lines 11 and 12 in Algorithm 1. Table 5.2 summarizes these strategies.

The **Stopped** strategy discards any subsequent data point when the memory limit is reached. It is the most straightforward method as it only uses the model created so far. In this strategy, functions `update_box` and `update_counters` are no-ops. The Stopped strategy has the advantage of not corrupting the node's box with outlier data points that would have required a split earlier in the tree. However, it also drops a lot of data after the model reached the memory limit.

Algorithm 1: Recursive function to train a node in the data stream Mondrian forest. This function is called on each root node of the forest when a new labeled data point arrives. The implementation of functions `update_box` and `update_counters` called at lines 11 and 12 vary depending on the out-of-memory strategy adopted (see Section 5.2.3). Table 5.2 summarizes these strategies. Table 5.1 describes the attributes of the node data structure.

```
   Data: x = a data point
   Data: l = label of the data point
   Data: node = current node containing attributes in Table 5.1
1  Function train_node(x, l, node) is
2      m = is_enough_memory();
3      distance = node.distance_to_box(x);
4      if distance is positive then
5          r = random_bool(node, node.parent, distance);
6          if r and m then
7              extend_tree(node, node.parent, x, l);
8              return;
9          end
10     end
11     update_box(node, x, r, m);
12     update_counters(node, l, r, m);
13     if !node.is_leaf() then
14         if x[node.split_feature] < node.split_value then
15             train_node(x, l, node.right);
16         else
17             train_node(x, l, node.left);
18         end
19     end
20 end
```

Attributes	Description
parent	Node's parent or empty for the root.
split_feature	Feature used for the split.
split_value	Value used for the split.
right	Right child of the node.
left	Left child of the node.
counters	An array counting labels
lower_bound	An array saving the minimum value for each feature
upper_bound	An array saving the maximum value for each feature
prev_lower_bound	Same as lower_bound but saving the previous minimum
prev_upper_bound	Same as upper_bound but saving the previous maximum

Table 5.1: Summary of the node structure used in Algorithm 1.

The **Extend Node** strategy disables the creation of new nodes when the memory limit is reached. Each data point is passed down the tree and no splits are created. However, the counters and the box of each node are updated. In Algorithm 1, functions `update_box` and `update_counters` work as if there were no split, thus updating counters and nodes as if `r` was always false. Compared to the Stopped method, the Extend Node one includes all the data points in the model. However, since the tree structure does not change, outlier data points may extensively increase node's boxes and as a result Mondrian trees tend to have large boxes, which is 1) detrimental to classification performance, and 2) limits further node creations in the event that more memory becomes available since the distance to the node's box is unlikely to be positive when computing m at line 4 of Algorithm 1.

The **Partial Update** strategy discards the points that would have created a split if enough memory was available, and updates the model with the other ones. This strategy discards fewer data points than in the Stopped method while it is less sensitive to outlier data points than the Extend Node method. In Algorithm 1, functions `update_box` and `update_counters` update statistics as in the original implementation, except when a split is triggered ($r = true$). In that case, all modifications done previously are canceled. Algorithm 2 and 3 describe functions `update_counters` and `update_box` in more details.

The **Count Only** strategy never updates node boxes and simply updates the counters with every new data point. In Algorithm 1, the function `update_box` is a no-op, while the

Method	update_counters	update_box
Stopped	—	—
Extend Node	Always	Always
Partial Update	Only if no split is triggered	Only if no split is triggered
Count Only	Always	—
Ghost	Update until a split is triggered	Update until a split is triggered

Table 5.2: Summary of the proposed out-of-memory strategies. `update_counters` and `update_box` refer to the functions in Algorithm 1. —: no-op.

function `update_counters` works as in the original implementation. This strategy is less sensitive to outliers than the Extend Node method and it discards less data points than Partial Update. However, it may create nodes that count data points outside of their box, thus nodes that do not properly describe the data distribution.

The **Ghost** strategy is similar to Partial Update for data points that do not create a split ($r = false$). In case of a split ($r = true$), the data point is dropped, however, in contrast with the Partial Update method, the changes applied between the node parent and the root are not canceled. This allows internal nodes to keep some information about data points that would have introduced a split, which preserves some information from the discarded data points.

5.2.4 Concept Drift Adaptation for Mondrian Forest under Memory Constraint

In this section, we propose methods to adapt Mondrian forests to concept drifts under memory constraints. We design and compare mechanisms to free up some memory by trimming tree leaves, and resume the growth of the forest after an out-of-memory strategy was applied. More specifically, we propose three methods to select a tree leaf for trimming: (1) Random trimming, (2) Data Count, and (3) Fading Count. In all cases, the trimming mechanism is called periodically on all trees when the memory limit is reached.

The **Random** method, used as baseline, selects a leaf to trim with uniform probability. The **Count** method selects the leaf with the lowest data point counter: it assumes that leaves with few data points are less critical for the classification than leaves with many data points. The exception to this would be leaves that have been recently created. To address

Algorithm 2: Partial Update algorithm for function `update_counters`.

Data: node = current node containing attributes Table 5.1

Data: l = label of the data point

Data: r = true if a split has been introduced.

Data: m = true if there is enough memory to run extend_tree.

```
1 Function update_counters(node, l, r, m) is
2     node.counters[l] += 1;
3     if m is false and r is true then
4         while node.parent is not root do
5             node.counters[l] -= 1;
6             node = node.parent
7         end
8     end
9 end
```

this issue, the **Fading count** method applies a fading factor to the leaf counters: when a new data point arrives in a leaf, the leaf counter c is updated to $c = 1 + c \times f$, where f is the fading factor. The other leaves get their counter updated to $c = c \times f$. As a result, leaves that haven't received data points recently are more prone to be discarded.

For all methods, the selected leaf is not trimmed if it contains more data points than a configurable threshold. This threshold prevents the trimming of an important leaf. The trimming mechanism is triggered every hundred data points when the memory limit is reached.

Once leaves have been trimmed, new nodes are available for the forest to resume its growth. The forest can extend according to the original algorithm, meaning that only new data points with features outside of the node boxes can create a split. However, with the Extend Node and Ghost strategies, this method creates leaves containing mostly outliers since node boxes have extensively increased when memory was full. This scenario is problematic because these new leaves might be even less used than the trimmed ones.

To address this issue, we propose to split the tree leaves that may have expanded as a result of the memory limitation in the Extend Node and Ghost strategies. The splitting of tree leaves is defined from two points: a new data point, and a split helper (Figure 5.1). The split is triggered in the tree leaf that contains the new data point, along a dimension defined using the split helper. We propose two variants of the splitting method corresponding to different definitions of the split helpers: in variant **Split AVG**, the split helper is the fading

Algorithm 3: Partial Update algorithm for function `update_box`.

Data: node = current node containing attributes Table 5.1

Data: x = the data point

Data: r = true if a split has been introduced.

Data: m = true if there is enough memory to run extend_tree.

```
1  Function update_box(node, x, r, m) is
2      foreach f ∈ all features do
3          node.prev_lower_bound[f] = node.lower_bound[f];
4          node.prev_upper_bound[f] = node.upper_bound[f];
5          node.lower_bound[f] = min(x[f], node.lower_bound[f]);
6          node.upper_bound[f] = min(x[f], node.upper_bound[f]);
7      end
8      if m is false and r is true then
9          while node.parent is not root do
10             foreach f ∈ all features do
11                 node.lower_bound[f] = node.prev_lower_bound[f];
12                 node.upper_bound[f] = node.prev_upper_bound[f];
13             end
14             node = node.parent
15         end
16     end
17 end
```

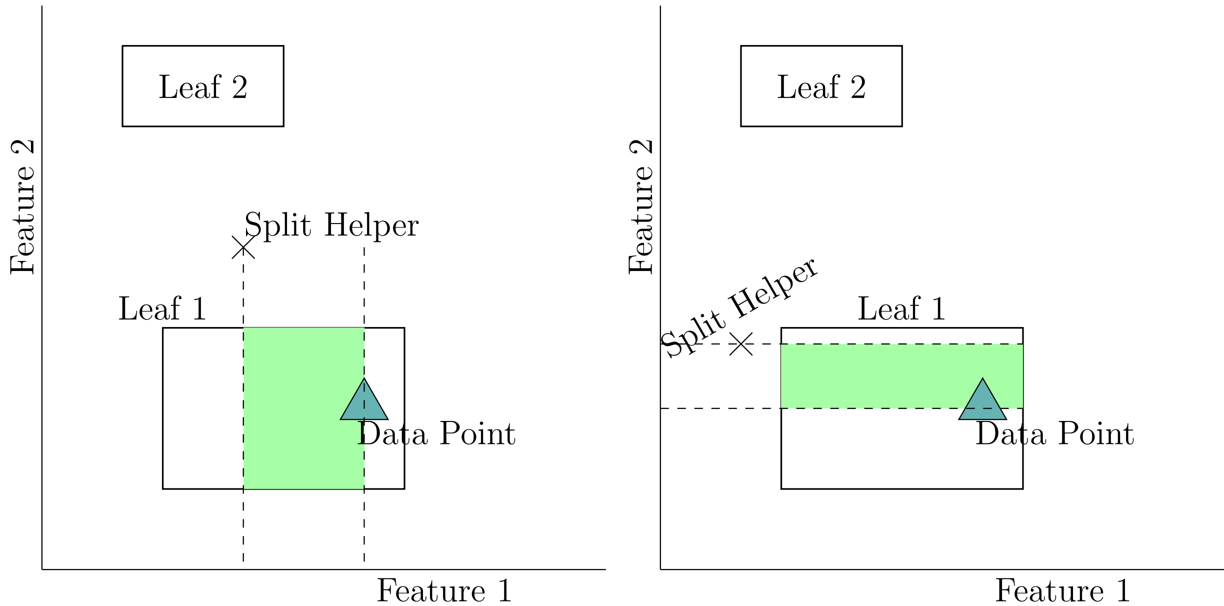


Figure 5.1: A two-feature example of the split definition. In each scenario, rectangles represent leaf boxes, the triangle is the new data point, and the cross is the split helper. The green area indicates the range of values where the new split will be created. The split helper is arbitrarily placed to illustrate different situations.

average data point, whereas in **Split Barycenter**, it is the weighted average of the leaves. The split dimension is randomly picked amongst the dimensions along which the split helper features are included within the leaf box. The split value along this dimension (green region in Figure 5.1) is randomly picked between the data point feature value and the split helper feature value along this dimension. The counters in the original leaf are proportionally split between the new leaves. The idea behind the split helper is to create new leaves in the parts of the tree that already contain a lot of data points.

5.2.5 Time Complexity

In this section, we discuss the time complexity of the training and testing processes. Equations 1 and 2 respectively describe the time complexity for the *training* and *testing* processes, where:

- m is the memory size in number of nodes.
- t is the tree count.
- d is the depth of the tree $\simeq log(\frac{m}{t})$ in a case of a balanced tree.

- l is the label count.
- f is the feature count.

$$training = \mathcal{O}(td(f + l)) \tag{1}$$

$$testing = \mathcal{O}(ml + tl + td(f + l) + l) \tag{2}$$

Indeed, in the training process, the t term is related to training each tree of the ensemble. The data point is sorted into a leaf, which gives the d term. Finally, at each depth level, the process computes distances, and updates the node's bounds and the node counters, which gives us the $(f + l)$ terms.

For the testing process, the model starts by updating statistics in each node for the labels, which gives ml in Equation 2. Then it processes the score of each tree, which gives the tl term. Similarly to the training process, the predict process will sort the data point into a leaf while computing distances and nodes score, which adds the term $td(f + l)$. Finally, the tree scores are aggregated and it adds l.

We note that, despite being constant in regards to the stream size, the time complexity is impacted by dataset characteristics (label count and feature count) as well as user-defined parameters (memory size and tree count). We also note that for all variables included in Equation 1 and 2, none of them expanded into quadratic terms.

The out-of-memory strategies do not influence these equations, however, the trimming and split methods add a few terms to the training process. The Trim Fading method needs to fade the count of all leaves, which adds ml as shown in Equation 3.

$$training_trim_fading = \mathcal{O}(ml + td(f + l)) \tag{3}$$

Besides, the Split AVG has to maintain an average data point, which introduces the term f and gives Equation 4.

$$training_split_AVG = \mathcal{O}(f + td(f + l)) \tag{4}$$

5.2.6 Node Boxes Analysis

In this section, we analyze the impact of node boxe sizes on the quality of tree predictions based on the Mondrian forest algorithm provided in [71].

In the classification step, the Mondrian tree passes the data point x through the tree, and computes a score S_k for each label k. The prediction of the forest is the label with the highest score S_k.

The score S_k mainly depends on the following terms:

- G_j, the predictive probability at node j.
- $p_j(x)$ the probability for data point x to generate a split at node j.
- $P_{NotSperatedYet}(j,x)$ the probability of not having generated a split yet as node j.
- $s_{j,k}(x)$ the score given by a node j to assign label k to data point x.
- $\eta_j(x)$ the distance to the node's box.

The following paragraphs explain how S_k is computed.

$\eta_j(x)$ is computed as follows:

$$\eta_j(x) = \sum_d (max(x_d - u_{jd}, 0) + max(l_{jd} - x_d, 0)), \tag{5}$$

where:

- u_{jd} and l_{jd} are respectively the upper and lower bound at node j on feature d,
- x_d is the value of the data point x for feature d.

$\eta_j(x)$ depends on the distance between x and the box of node j defined by u_j and l_j for all features. If x falls within the box of j, $\eta_j(x)$ will be zero, and the farther x is from j's box, the higher $\eta_j(x)$ will be.

Equation 6 shows how $p_j(x)$ is computed based on $\eta_j(x)$ and Δ_j, a distance between node j and its parent. We note that when x falls within the box of j, $\eta_j(x)$ is equal to zero, and thus, $p_j(x)$ is null.

$$p_j(x) = 1 - exp(-\Delta_j \eta_j(x)) \tag{6}$$

Equation 7 defines $P_{NotSperatedYet}(j)$, the probability of not having branched off before reaching node j. It uses $p_j(x)$, the probability to branch off at node j defined in Equation 6, and $ancestors(j)$ which returns the ancestors of node j starting from the root.

$$P_{NotSperatedYet}(j,x) = \prod_{g \in ancestors(j)} (1 - p_g(x)) \tag{7}$$

Equation 8 describes how the score of node j is computed for label k. It uses $p_j(x)$, the probability of branching off at that node, $P_{NotSperatedYet}(j)$, defined in Equation 7, and $G_{j,k}$, the predictive probability at node j for label k.

$$s_{j,k}(x) = \begin{cases} P_{NotSperatedYet}(j,x)(1-p_j(x))G_{j,k} & j = \text{leaf.} \\ P_{NotSperatedYet}(j,x)p_j(x)G_{j,k} & \text{otherwise.} \end{cases} \tag{8}$$

The score given by a tree for label k and data point x, $S_k(x)$, is shown in Equation 9. $leaf(x)$ returns the leaf node where the data point x has been sorted to. $path(j)$ returns the list of nodes that lead to node j, starting from the root.

$$S_k(x) = \sum_{j \in path(leaf(x))} s_{j,k}(x) \tag{9}$$

We can see from the computation of S_k that the strategies that do not expand the node boxes (Count Only and Stopped, as well as Partial Update and Ghost to a certain extent) tend to use the tree root rather than leaves to predict class labels, even though the root only has a very rough approximation of class distributions. Indeed, in a situation where boxes are maintained small such as with Count Only strategy, the data point will have a greater chance to fall outside a node box as well as farther from it. In this case, the probability of branching off $p_j(x)$, will be higher and thus $P_{NotSperatedYet}(j)$ will be lower as we go deeper in the tree.

Therefore, when computing $S_k(x)$, the node score $s_{j,k}$ will become smaller as we get closer to a leaf because $s_{j,k}$ is the product of the node prediction $G_{j,k}$ weighted by $P_{NotSperatedYet}(j,x)$. In that situation, most of the score comes from nodes closer to the root, but these nodes have a poor approximation of the feature space.

For nodes closer to the leaf, analysis shows that their weight in the tree prediction increases as box sizes increase. In general, with a smooth label distribution, we think that Extend Node would exhibit better performance for this reason. We also expect these nodes to become irrelevant in a concept drift situation. In this scenario, using nodes closer to the root may be more relevant.

The following empirical study intends to compare the different approaches and determine if it is better to increase the impurity of the node by forcing the box to extend (Extend Node strategy) and include data points that should have branched off than keeping the box small and having a finer-grain partition of the space (Count Only, Stopped, Partial Update, and Ghost).

5.2.7 Datasets

We used six datasets to evaluate our proposed methods: three synthetic datasets to mimic real-world situations and to make comparisons with and without concept drifts, and two real Human Activity Recognition datasets.

Banos *et al*

The Banos *et al* dataset [9][1] is a human activity dataset with 17 participants and 9 sensors per participant. Each sensor samples a 3D acceleration, gyroscope, and magnetic field, as well as the orientation in a quaternion format, producing a total of 13 values. Sensors are sampled at 50 Hz, and each sample is associated with one of 33 activities. In addition to the 33 activities, an extra activity labeled 0 indicates no specific activity.

We pre-processed the Banos *et al* dataset as in [13], using non-overlapping windows of one second (50 samples), and using only the 6 axes (acceleration and gyroscope) of the right forearm sensor. We computed the average and standard deviation over the window as features for each axis,similar to the second feature set in [13]. Indeed, this feature set was providing the best performance while minimizing the producing cost.. We assigned the most frequent label to the window. The resulting data points were shuffled uniformly.

In addition, we constructed another dataset from Banos *et al*, in which we simulated a concept drift by shifting the activity labels in the second half of the data stream.

Recofit

The Recofit dataset [79, 80] is a human activity dataset containing 94 participants. Similar to the Banos *et al* dataset, the activity labeled 0 indicates no specific activity. Since many of these activities were similar, we merged some of them together using the same logic as in Table 4.1.

We pre-processed the dataset similarly to the Banos *et al* dataset, using non-overlapping windows of one second, and only using 6 axes (acceleration and gyroscope) from one sensor. From these 6 axes, we used the average and the standard deviation over the window as features. We assigned the most frequent label to the window.

[1] available here

MOA Datasets

We generated two synthetic datasets using Massive Online Analysis [18] (MOA) is a Java framework to compare data stream classifiers. In addition to classification algorithms, MOA provides many tools to read and generate datasets. We generate two synthetic datasets (MOA commands available here) using the RandomRBF algorithm, a stable dataset, and a dataset with a drift. Both datasets have 12 features and 33 labels, similar to the Banos *et al* dataset. We generated 20,000 data points for each of these synthetic datasets.

Covtype

The Covtype dataset[2] is a tree dataset. Each data point is a tree described by 54 features including ten quantitative variables and 44 binary variables. The 581,012 data points are labeled with one of the seven forest cover types and these labels are highly imbalanced. In particular, two labels represent 85% of the dataset.

5.2.8 Evaluation Metric

We evaluated our methods using a prequential fading macro F1-score. We used a prequential metric because data stream models cannot be evaluated with the traditional testing/training sets since the model continuously learns from a stream of data points [43]. We focused on the F1 score because most datasets are imbalanced. We used the prequential version of the F1 score to evaluate classification on data stream. Contrary to the previous chapter, we used a fading factor to minimize the impact of old data points, especially data points at the beginning or data points saw before a drift occur. To obtain this fading F1 score, we multiplied the confusion matrix with the fading factor before incrementing the cell in the confusion matrix.

5.3 Results

Our experiments evaluated the out-of-memory and adaptation strategies presented previously. We evaluated both sets of methods separately. Unless specified otherwise, we allocated 600 KB of memory for the forest, which allowed for 940 to 1600 nodes in the forest depending on the number of features, labels, and trees. As a comparison, the Raspberry Pi Pico has 256 KB of memory and the Arduino has between 2 KB (Uno) and 8 KB (MEGA 2560).

[2] available here

5.3.1 Baselines

Figure 5.2 shows the F1 score obtained at the end of each dataset as a function of the number of trees in the forest. The experiment includes the methods described in Section 5.2.3 — executed wih a memory limit of 0.6 MB — as well as the online Mondrian forest and the Data Stream Mondrian forest with 2 GB of memory for reference. The limit of 2 GB is reached for the Covtype and Recofit datasets, and the Data Stream Mondrian 2 GB method applies the extend node method in that situation. The Online Mondrian is the Python implementation available on GitHub [12] that we modified to output the F1 score.

The Online Mondrian reaches state-of-the-art performances in the real datasets [21, 78, 26, 31]. The Data Stream Mondrian 2 GB achieves similar performances for the Banos *et al* dataset and RandomRBF stable. However it is significativly lower with the Recofit dataset, the RandomRBF with drift, and the Covtype dataset. These differences are explained by three factors: the 2 GB limit is reached for the Recofit and Covtype datasets; the Online Mondrian is evaluated with a holdout set randomly selected from the dataset whereas the Data Stream Mondrian is evaluated with a prequential method (as is commonly done in data streams); the Online Mondrian can access to previous data points while the Data Stream Mondrian cannot. The Online Mondrian is given here as a reference for comparison, however, it should not be directly compared to the Data Stream Mondrian as these methods operate in different contexts.

The Stopped method, the default reference for evaluation under memory constraints, has by far the lowest F1 score, which demonstrates the usefulness of our out-of-memory strategies. The impact of memory limitation is clear and can be seen by the substantial performance edge of Data Stream Mondrian 2 GB over the other methods.

5.3.2 Out-of-memory strategies

Figure 5.2 shows that for the stable and drift RandomRBF datasets all our proposed out-of-memory strategies achieve similar F1 scores — clearly better than the default Stopped method. Since none of these methods take concept drifts into account, the RandomRBF with drift exhibit F1 scores lower than RandomRBF stable by roughly 0.65.

In the four other datasets, the Extend Node method consistently stands above the other ones except for the Banos *et al* with drift dataset where the Partial Update method achieves the best F1 score. This is due to a faster recovery of the nodes' counters after the drift. The other three methods, Count Only, Ghost, and Partial Update, tend to have similar F1

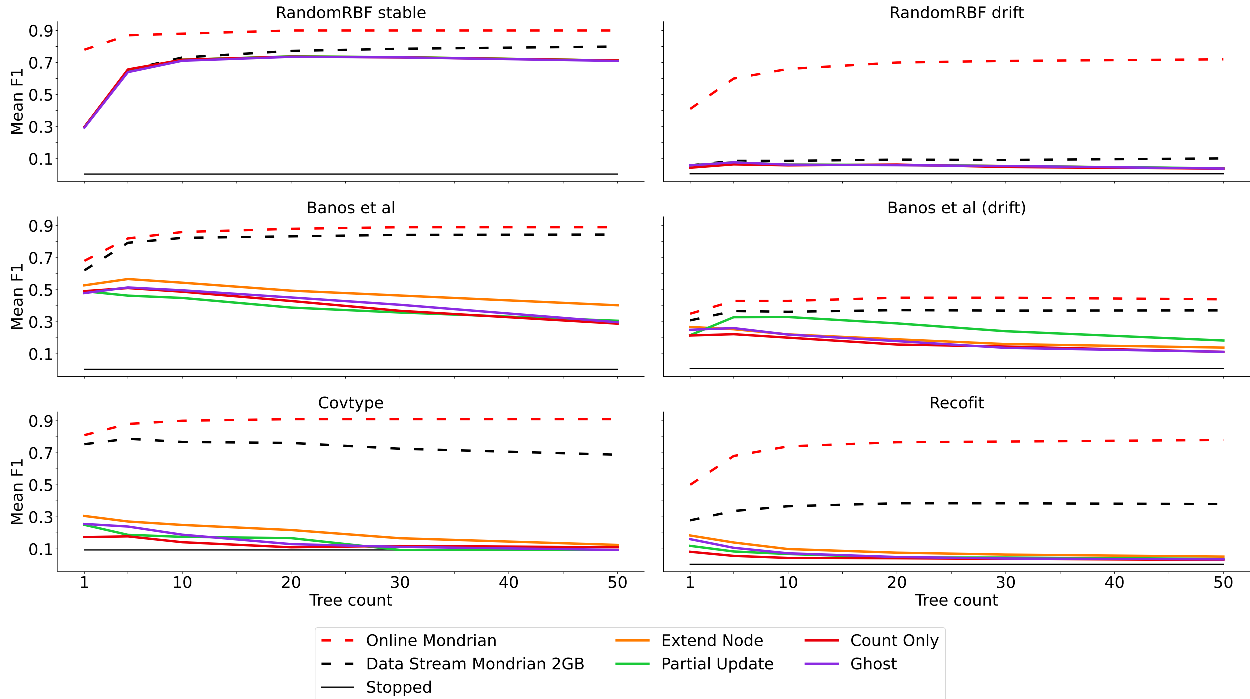


Figure 5.2: Comparison of the out-of-memory strategies proposed in Section 5.2.3 for six datasets.

scores. The Extend Node makes the best of two factors. First, it does not drop any data points compare to Partial Update or Count Only. Second, since it extends the boxes, future data points fall within the box and receive the prediction of the corresponding leaf, whereas the Count Only and the Ghost methods soften the leaf prediction depending on the data point distance with the box.

The difference in F1 score between the Stopped Mondrian and the other methods is reported in Table 5.3. The difference is computed for all numbers of trees and we report the minimum, the mean, and the maximum. On average, the Extend Node has the best improvement over Stopped Mondrian with an average improvement in F1-score of 0.27, a minimum of 0.03, and a maximum of 0.73.

Finally, we note on Figure 5.2 that the F1 score generally decreases when the number of trees increases. This deterioration occurs due to the trade-off between the number of trees and the trees' size. Indeed, since the memory bound is shared by all trees, the more trees there are, the shallower they must be, making them more prone to underfit. This does not happen when enough memory is available because in such cases, trees do not influence each other's size. This is why the Mondrian 2GB reaches plateaus instead of decreasing. Therefore, under memory constraint, the number of trees has a significant impact on the

Dataset	Method Name	Δ F1 Min	Mean	Max
RandomRBF stable	Count Only	**.29**	**.64**	**.73**
	Extend Node	**.29**	**.64**	**.73**
	Ghost	**.29**	.63	**.73**
	Partial Update	**.29**	**.64**	**.73**
RandomRBF drift	Count Only	**.03**	**.05**	.06
	Extend Node	**.03**	**.05**	**.07**
	Ghost	**.03**	**.05**	**.07**
	Partial Update	**.03**	**.05**	**.07**
Banos et al	Count Only	.28	.43	.51
	Extend Node	**.40**	**.50**	**.56**
	Ghost	.29	.44	.51
	Partial Update	.30	.41	.49
Banos et al (drift)	Count Only	.10	.17	.21
	Extend Node	.13	.20	.26
	Ghost	.10	.18	.25
	Partial Update	**.17**	**.26**	**.32**
Covtype	Count Only	.02	.04	.08
	Extend Node	**.03**	**.13**	**.21**
	Ghost	.00	.08	.16
	Partial Update	.00	.07	.16
Recofit	Count Only	.03	.04	.08
	Extend Node	**.05**	**.10**	**.18**
	Ghost	.03	.07	.16
	Partial Update	.03	.06	.11

Table 5.3: ΔF1 score compared to Stopped Mondrian. Minimum, maximum, and average scores are computed across all tree numbers.

performance.

From these results, we conclude that the Extend Node should be the default strategy to follow when the Mondrian forest reaches the memory limit.

5.3.3 Concept Drift Adaptation for Mondrian Forest under Memory Constraint

Figure 5.3 shows the F1 score for the three proposed leaf trimming strategies. All trimming strategies were evaluated with the Extend Node out-of-memory strategy as it outperforms the other out-of-memory strategies per our previous experiment. In the case of concept drift (RandomRBF drift, Banos *et al* drift), the random trimming method outperforms the other ones whereas for the stable datasets (RandomRBF, Banos *et al*, Covtype, Recofit), no trimming appears to be the best strategy.

Figure 5.4 shows the F1 score for the three splitting methods (no split, Split AVG, and Split Barycenter) combined with random trimming and extend node as the out-of-memory strategy. With the two drift datasets, the three methods perform better than not trimming. With RandomRBF stable, the two splitting methods perform better than not trimming with 5 and 10 trees. Finally, with Banos *et al*, Recofit, and Covtype datasets, not trimming is

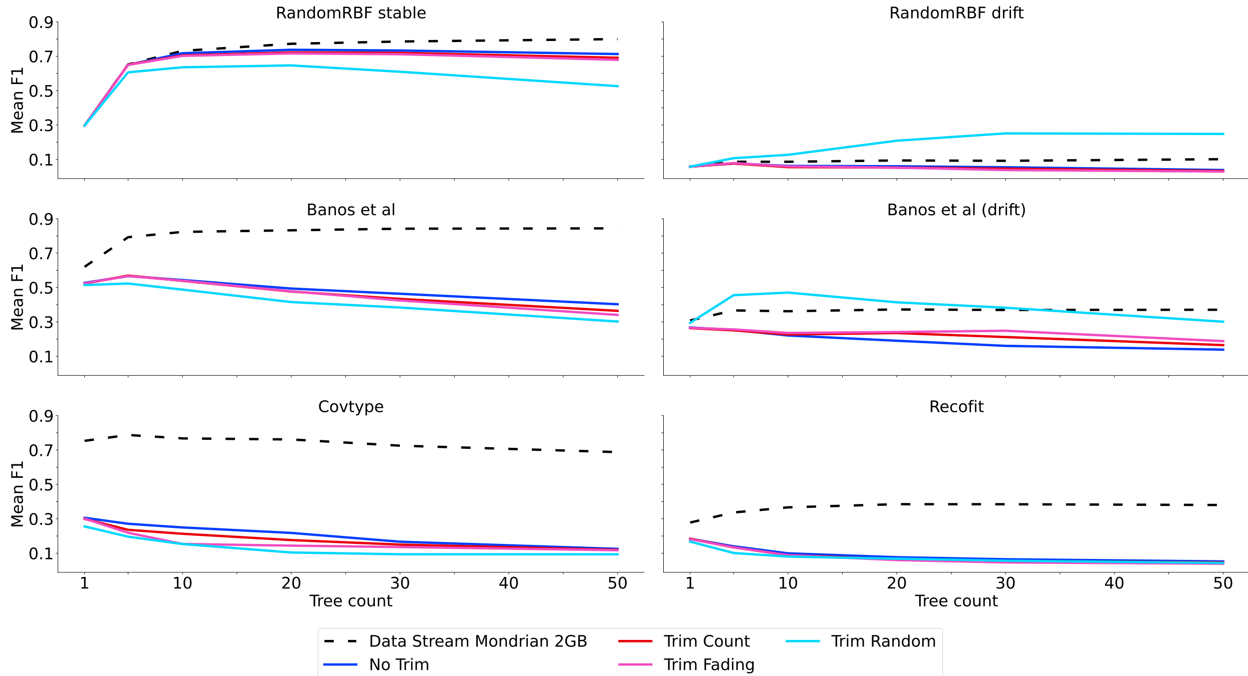


Figure 5.3: Comparison of trimming methods applied with the Extend Node out-of-memory strategy.

generally better than randomly trimming.

On Figure 5.4, we note that for the RandomRBF stable with 5 and 10 trees, the split methods get their F1 score higher than No Trim and Mondrian 2GB. Their F1 scores improve over time even though the dataset is stable.

Table 5.4 summarizes the delta of F1 scores of the trimming methods compared to the Stopped method. From these results, we conclude that randomly trimming allows the Mondrian forest to adapt to concept drift. We also conclude that using the split methods (Split AVG and Split Barycenter) should be the default method to grow the Mondrian trees after trimming as they exhibit a better F1 score than not splitting.

5.3.4 Impact of the Memory Limit

We note in Figures 5.2, 5.3, and 5.4, that the F1-scores tend to be low even though the Data Stream Mondrian 2 GB reaches state-of-the-art F1-scores. This implies that reducing the memory from 2 GB to 600 KB has a strong impact on the performance. We also note that this impact varies between datasets. For the RandomRBF stable dataset, the methods are closer to the Data Stream Mondrian 2 GB compared to the Covtype dataset where there is a more important difference.

Dataset	Method Name	Δ F1 Min	Mean	Max
RandomRBF stable	Trim Count, No Split	**.29**	.63	.72
	Trim Count, Split AVG	**.29**	**.66**	.74
	Trim Count, Split Barycenter	**.29**	**.66**	.75
	Trim Fading, No Split	**.29**	.62	.71
	Trim Fading, Split AVG	**.29**	**.66**	**.76**
	Trim Fading, Split Barycenter	**.29**	**.66**	**.76**
	Trim Random, No Split	**.29**	.55	.64
	Trim Random, Split AVG	**.29**	.63	.74
	Trim Random, Split Barycenter	**.29**	.62	.73
RandomRBF drift	Trim Count, No Split	.03	.05	.07
	Trim Count, Split AVG	**.05**	.09	.13
	Trim Count, Split Barycenter	**.05**	.09	.14
	Trim Fading, No Split	.02	.05	.07
	Trim Fading, Split AVG	**.05**	.25	.32
	Trim Fading, Split Barycenter	**.05**	.25	.33
	Trim Random, No Split	**.05**	.16	.25
	Trim Random, Split AVG	**.05**	**.27**	**.38**
	Trim Random, Split Barycenter	**.05**	.26	.37
Banos et al	Trim Count, No Split	.36	.48	**.57**
	Trim Count, Split AVG	**.38**	**.49**	.56
	Trim Count, Split Barycenter	**.38**	**.49**	**.57**
	Trim Fading, No Split	.34	.48	.56
	Trim Fading, Split AVG	.36	.48	.56
	Trim Fading, Split Barycenter	.37	.48	.56
	Trim Random, No Split	.30	.43	.52
	Trim Random, Split AVG	.25	.43	.54
	Trim Random, Split Barycenter	.26	.43	.53
Banos et al (drift)	Trim Count, No Split	.16	.22	.25
	Trim Count, Split AVG	.20	.28	.32
	Trim Count, Split Barycenter	.20	.28	.31
	Trim Fading, No Split	.18	.23	.26
	Trim Fading, Split AVG	.27	.38	.46
	Trim Fading, Split Barycenter	.28	**.39**	.47
	Trim Random, No Split	**.29**	.38	.46
	Trim Random, Split AVG	.23	.37	**.48**
	Trim Random, Split Barycenter	.24	.37	.47
Covtype	Trim Count, No Split	.03	.10	**.21**
	Trim Count, Split AVG	.03	**.12**	**.21**
	Trim Count, Split Barycenter	.03	**.12**	**.21**
	Trim Fading, No Split	.02	.08	**.21**
	Trim Fading, Split AVG	**.04**	**.12**	**.21**
	Trim Fading, Split Barycenter	**.04**	**.12**	.20
	Trim Random, No Split	.00	.06	.16
	Trim Random, Split AVG	.02	.07	**.21**
	Trim Random, Split Barycenter	.02	.06	.18
Recofit	Trim Count, No Split	.04	.09	.18
	Trim Count, Split AVG	**.05**	**.10**	.18
	Trim Count, Split Barycenter	**.05**	**.10**	.18
	Trim Fading, No Split	.03	.09	.18
	Trim Fading, Split AVG	**.05**	**.10**	**.19**
	Trim Fading, Split Barycenter	.04	**.10**	**.19**
	Trim Random, No Split	.04	.08	.16
	Trim Random, Split AVG	.03	.08	.17
	Trim Random, Split Barycenter	.03	.08	.17

Table 5.4: ΔF1 score compared to Stopped Mondrian for the trimming methods. Minimum, maximum, and average scores are computed across all tree numbers.

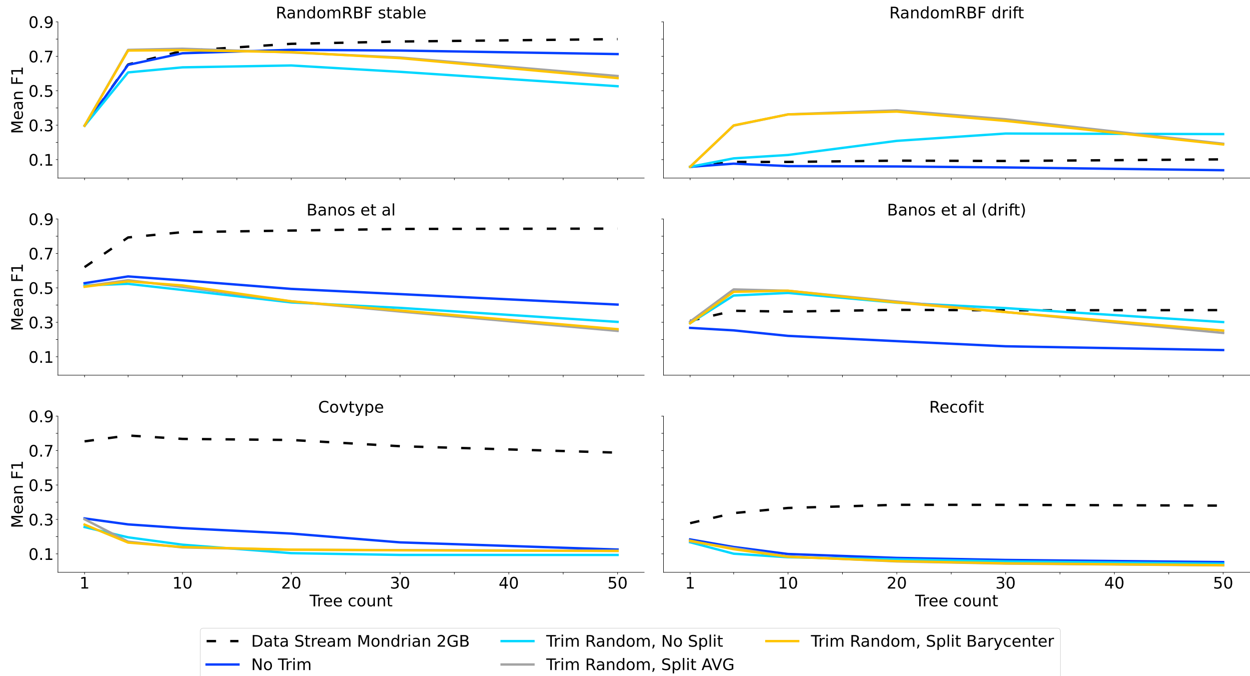


Figure 5.4: Comparison of tree leaf splitting methods combined with the Random trimming strategy.

Figure 5.5 shows the evolution of the memory limit impact on the F1-score for a Mondrian forest with 50 trees. We selected 50 trees as it is the number of trees that benefit the most from more memory. Indeed, more memory means they are less likely to underfit and thus, have better performance than fewer trees. Similar trends are observed with fewer trees. The dashed black line indicates the F1-score reached by Data Stream Mondrian 2 GB.

We note that, except for the Stopped method, F1-scores increase with the amount of available memory. This is explained by the fact that trees can grow more nodes and therefore describe a finer-grained partition of the feature space.

We observe that the sharpest improvement for the No Trim method happens between 600 KB and 10 MB, after which the F1-score increase slows down and plateaus toward the Data Stream Mondrian 2 GB F1-score.

The trimming methods with split exhibit two behaviors. For stable datasets, they follow the trend of No Trim. For the drift datasets, the trimming methods improve beyond the Data Stream Mondrian 2 GB up to the 10 MB limit, after which the F1-score decreases down to the Data Stream Mondrian 2 GB.

This behavior is explained by the trimming algorithm that trims based on the tree count rather than the tree size. When too much memory is allocated the size of the trees becomes

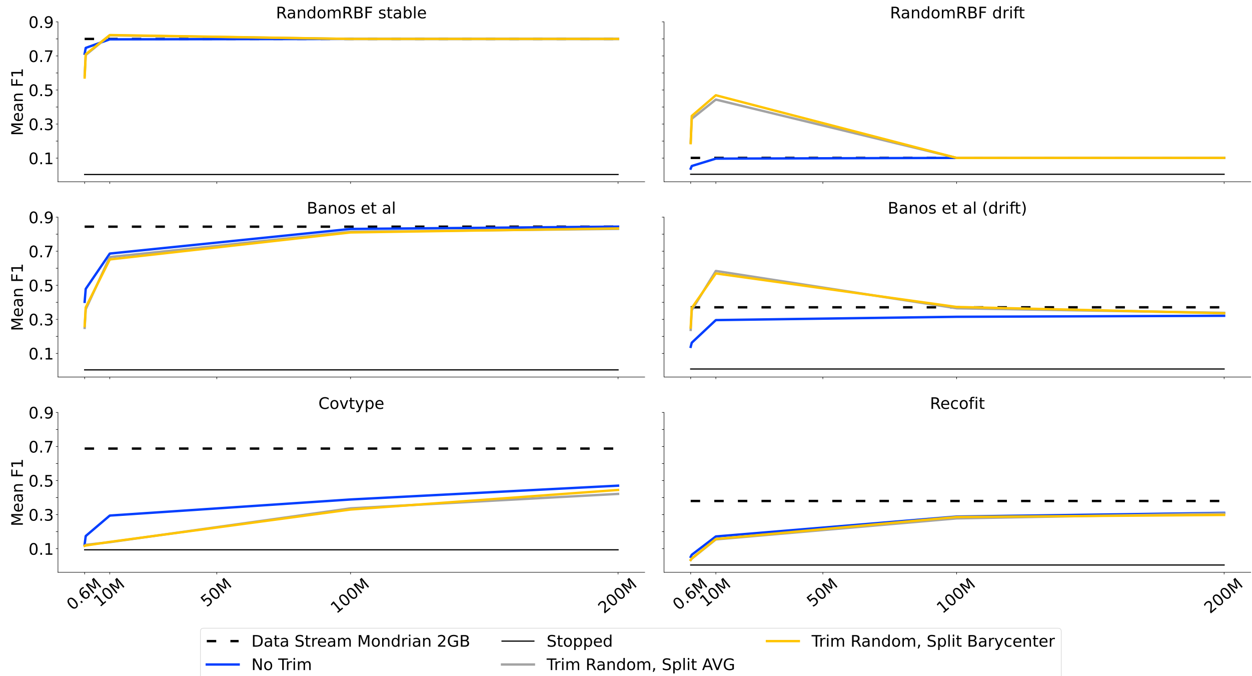


Figure 5.5: Evaluation of the memory impact on the top out-of-memory strategy and the top trimming methods described in Section 5.2.3 and Section 5.2.4. The results are shown for 50 trees.

too big for the trimming pace. Therefore, the old concept is not trimmed out fast enough, which explains the decrease in F1-score beyond the 10 MB mark for the trimming methods on the drift datasets.

5.4 Related Work

Edge computing is a concept where processing is done close to the device that produced the data, which generally means on devices with much less memory than regular computing servers. There are many surveys about classification for edge computing [81, 91, 37], but most of the work focuses on deep learning, which is not applicable in our case because it requires a lot of data and time to train the model. They discuss inherent problems related to learning with edge devices, in particular about lighter architecture and distributed training. They also depict areas where machine learning on edge devices would be impactful like computer vision, fraud detection, or autonomous vehicles. Finally, these studies draw future work opportunities such as data augmentation, distributed training, and explainable AI. Aside from the deep learning approaches, the survey in [81] discusses two machine learning

techniques with a small memory footprint: the Bonsai and ProtoNN methods.

Bonsai [70] is a tree-based algorithm designed to fit in an edge device memory. It is a sparse tree that comes with a low-dimension projection of the feature space to improve learning while limiting memory usage and achieving state-of-the-art accuracy. Similarly, ProtoNN [51] is a kNN based model that performs a low-dimension projection of the features to increase accuracy and improve its memory footprint. It also compresses the training set into a fixed amount of clusters. ProtoNN and Bonsai claim to remain below 2KB while retaining high accuracy. However, these models don't apply to evolving data streams because their low-dimension projections and their structures are pre-trained based on existing data, thus, adjusting them would require more time and memory. Additionally, our method starts from scratch whereas ProtoNN or Bonsai require data before being used.

When it comes to forests designed for concept drift, there are many variations and many mechanisms. The Hoeffding Adaptive Tree [16] embeds a concept drift detector and grows a ghost branch when it detects a drift in a branch. The ghost branch replaces the old one when its performance becomes better. Similarly, the Adaptive Random Forest [17] keeps a drift detector for each tree and starts growing a ghost tree when a drift is detected. The work in [19] presents a forest that is constantly evolving where each decision tree has its size limit. When the limit is reached, it restarts from the last created node. With this mechanism, trees with a smaller limit will adapt faster to recent data points. Additionally, the forest also embeds the ADWIN [5] concept drift detector and restarts the worst base learner when a drift is detected. This mechanism called the ADWIN bagging is used in the Adaptive Rotation Forest [102] in addition to a low-dimension projection of the features with an incremental PCA. Such combinations allow the forest to maintain the most accurate base classifiers while keeping the projection up-to-date. Our methods differ from ADWIN or the Adaptive Rotation Forest because we rely on passive drift adaptation rather than using a drift detector.

The Kappa Updated Ensemble [26] is an ensemble method that notices drifts by self-monitoring the performance of its base classifiers. In case of a drift, the model trains new classifiers. The prediction is made using only the best classifiers from the ensemble but the method never discards a base classifier as it can still be useful in the future. This mechanism of keeping unused trees raises a memory problem since it may fill the memory faster for very little benefit.

Similarly, the work in [38] proposes a method to prune base learners based on their global and class-wise performances. It is used to reduce memory consumption while retaining good

accuracy across all classes. The method evaluates the base learners for each class then ranks them using a modified version Borda Count.

The Mean error rate Weighted Online Boosting [55] is an online boosting method where the weights are calculated based on the accuracy of previous data. Even though the method is not designed for concept drift, the self-monitoring of the accuracy makes the base learner train more on recent data making the ensemble robust to concept drift.

These last studies rely on ranking the base learners of the ensemble to either adjust or disable them. However, these coarse-grain approaches are memory intensive and are not applicable to the trimming methods because they would require keeping statistics for each node.

5.5 Conclusion

We adapted Mondrian forests to support data streams and we proposed five out-of-memory strategies to deploy them under memory constraints. Results show that the Extend Node method has the best improvement on average. With a carefully tuned number of trees, the Extend Node also has the highest F1 score gain compared to the Stopped strategy. Thus, we recommend using Extend Node as the default strategy.

We also compared node trimming methods for the Mondrian trees and there are two viable methods depending on the situation. Not trimming is the best option in most case of stable dataset. However, when expecting a concept drift, the trim Random with splits is preferable. The drawback with the Trim Random method is that it deteriorates the F1 score on stable streams.

Overall, this paper showed that using our out-of-memory strategies is critical in order to make the Mondrian forest work in a memory-constrained environment. In particular, existing Mondrian forest implementations [12, 54] do not have any out-of-memory strategy and will fail if they cannot allocate any more nodes. Using the Extend Node strategy allows an average F1 score gain of 0.28 accross all datasets compared to not doing anything. Similarly, using Trim Random offers an average F1 score gain of 0.3.

Our results show no significant difference between the two types of node splitting strategy. By default, we would recommend using Split AVG because it is less compute-intensive.

We observe that with a carefully picked number of trees, using trimming can lead to improvement with RandomRBF stable. This statement needs more investigation to understand why it happened on one dataset and if similar behavior can be obtained with the other

datasets.

Future work will focus on the tree count adjustment as our results have shown its critical impact on the performances. It will also investigate trim mechanisms to adaptively trim depending on the memory limit and concept drift. Finally, we suggest exploring the use of drift detectors such as ADWIN [5] to switch between No Trim and Trim Random.

Chapter 6

Dynamic Ensemble Size Adjustment for Memory Constrained Mondrian Forest

Published as: Martin Khannouz and Tristan Glatard. Dynamic Ensemble Size Adjustment for Memory Constrained Mondrian Forest. In *2022 IEEE International Conference on Big Data (Big Data)*, pages 3358–3363, Osaka, Japan, dec 2022. IEEE Computer Society.

Supervised learning algorithms generally assume the availability of enough memory to store data models during the training and test phases. However, this assumption is unrealistic when data comes in the form of infinite data streams, or when learning algorithms are deployed on devices with reduced amounts of memory. Such memory constraints impact the model behavior and assumptions. In this paper, we show that under memory constraints, increasing the size of a tree-based ensemble classifier can worsen its performance. In particular, we experimentally show the existence of an optimal ensemble size for a memory-bounded Mondrian forest on data streams and we design an algorithm to guide the forest toward that optimal number by using an estimation of overfitting. We tested different variations for this algorithm on a variety of real and simulated datasets, and we conclude that our method can achieve up to 95% of the performance of an optimally-sized Mondrian forest for stable datasets, and can even outperform it for datasets with concept drifts. All our methods are implemented in the OrpailleCC open-source library and are ready to be used on embedded systems and connected objects.

6.1 Introduction

Supervised classification algorithms mostly assume the availability of abundant memory to store data and models. This is an issue when processing data streams — which are infinite sequences by definition — or when using memory-limited devices as is commonly the case in the Internet of Things. We focus on the Mondrian forest, a popular online classification method. Our ultimate goal is to optimize it for data streams under memory constraints to make it compatible with connected objects.

The Mondrian forest is a tree-based, ensemble, online learning method with comparable performance to offline Random Forest [71]. Previous experiments highlighted the Mondrian forest sensitivity to the ensemble size in a memory-constrained environment [62]. Indeed, introducing a memory limit to the ensemble also introduces a trade-off between underfitting and overfitting. On the one hand, low tree numbers make room for deeper trees and increase the risk of overfitting. On the other hand, large tree numbers constrain tree depth due to the memory limitation and increase the risk of underfitting. Depending on the memory limit and the data distribution, a given ensemble size may either overfit or underfit the dataset. The goal of this paper is to address this trade-off by adapting the ensemble size dynamically.

In summary, this paper makes the following contributions:

1. Highlight the existence of an optimal tree count in memory-constrained Mondrian forest;
2. Propose a dynamic method to optimize the tree count;
3. Compare this method to the Mondrian forest with a optimally-sized tree count.

6.2 Materials and Methods

All the methods presented in this section are implemented in the OrpailleCC framework [63]. The scripts to reproduce our experiments are available on GitHub at `https://github.com/big-data-lab-team/benchmark-har-data-stream`.

In this section, we start by presenting background on Mondrian Forests (6.2.1 and 6.2.2), then presents the main contribution of the paper, namely dynamically adjusting the ensemble size of a memory-constrained Mondrian forest (6.2.3, 6.2.4, 6.2.5, 6.2.6), then describes the experimental evaluation framework (6.2.7, 6.2.8).

6.2.1 Mondrian Forest

The Mondrian forest [71] is an ensemble method that aggregates Mondrian trees. Each tree recursively splits the feature space, similar to a regular decision tree. However, the feature used in the split and the value of the split are picked randomly. The probability to select a feature is proportional to its range, and the value for the split is uniformly selected in the range of the feature. In contrast with other decision trees, the Mondrian tree does not split leaves to introduce new nodes. Instead, it introduces a new parent and a sibling to the node where the split occurs. The original node and its descendant are not modified and no data point is moved to that new sibling besides the data points that initialized the split. This approach allows the Mondrian tree to introduce new branches to internal nodes. This training algorithm does not rely on labels to build the tree, however, each node maintains counters for each label seen. Therefore, labels can be delayed, but are needed before the prediction. In addition to the counters, each node keeps track of the range of its feature which represents a box containing all data points. A data point can create a new branch only if it is sorted to a node and falls outside of the node's box.

6.2.2 Mondrian Forest for Data Stream Classification

The implementation of Mondrian forest presented in [71, 12] is online because trees rely on potentially all the previously seen data points to grow new branches. To support data streams, the Mondrian forest has to access data points only once as the dataset is assumed to be infinite in size.

The work in [62] describes a Data Stream Mondrian forest with a memory bound. The ensemble grows trees from a shared pool of nodes and the trees are paused when there is no node left. This work also proposed out-memory strategies to keep updating the statistics for the trees without creating new branches. In particular, this work recommend using the Extend Node strategy when the memory is full, a strategy where statistics of the node boxes are automatically extended to fit all data points, and the counters automatically increased.

The attributes of a node are an array for counting labels that fall inside a leaf, and two arrays *lower_bound* and *upper_bound* that define a box of the node. The Extend Node strategy automatically increases the counter of the label in the leaf and automatically adjusts *lower_bound* and *upper_bound* so the new data point fits inside the box.

Having a shared pool of nodes for the ensemble has a direct impact on the number of trees. As mentioned before, having more trees limits the tree depth and may lead to underfitting,

whereas having less trees increases the risk of overfitting.

6.2.3 Dynamic Tree Count Optimization

Algorithm 4 describes the function that trains the forest with a new data point and dynamically adjusts the ensemble size. The main idea is to compare pre- and post-quential errors to decide whether or not to adjust the forest size.

Algorithm 4: Training function for a data stream Mondrian forest with a dynamic ensemble size.

```
   Data: f = a Mondrian forest
   Data: x = a data point
   Data: l = the label of data point x
1  Function train_forest(f, x, l) is
2      predicted_label = f.test(x);
3      prequential.update(predicted_label, l);
4      for i do
5          train_tree(i, x, l);
6      end
7      predicted_label = f.test(x);
8      postquential.update(predicted_label, l);
9      post_metric = postquential.metric();
10     pre_metric = prequential.metric();
11     if post_metric >~ pre_metric then
12         trim_trees(f);
13         add_tree(f);
14     end
15 end
```

The forest evaluates prequential statistics, meaning that it predicts the new data point label before using it for training (line 2-3). After training, the forest evaluates postquential statistics (lines 7-8). The prequential accuracy is widely used as accuracy approximation on a test set for data streams [43]. Here we introduce the postquential accuracy, where we test after training, to simulate the accuracy on a training set.

To find the number of trees that maximize prediction performance, we initialize the Mondrian forest with a single tree, therefore with a high risk of overfitting. The idea is to

test for overfitting by comparing prequential and postquential accuracies and add a tree in case the forest is deemed to overfit.

A model that overfits is defined as a model that performs significantly better on the training set than on the test set. Therefore the problem of detecting overfitting becomes a statistical testing problem between the training and test accuracies.

While overfitting is commonly detected by comparing the performance of the classifier on the training and test set, detecting underfitting for memory-constrained data stream models remains very challenging. Consistently, Algorithm 4 is written to only support tree addition. Should an underfitting criterion be available in the future, Algorithm 4 could easily be adapted to support tree removal.

Overall, Algorithm 4 can be adjusted with three components: the comparison test (line 11), the type of pre/postquential statistics used (line 3 and 8), and the method to add trees (line 12-13).

6.2.4 Comparison Test

In Algorithm 4, the update process determines if it needs to add a tree based on a comparison between the prequential and the postquential accuracies. If both accuracies are significantly different, the forest is deemed to overfit and thus, the algorithm adds a tree to compensate.

There are different methods to test the statistical difference between two accuracies and we experiment with four in this paper: the sum of variances, the t-test, the z-test, and the sum of standard deviations.

Notations for the following equations include: μ_{pre} and μ_{post} the mean of respectively the prequential and postquential accuracies, σ^2_{pre} and σ^2_{post} the variance of respectively the prequential and postquential accuracies, n the size of the sample, μ and σ^2 respectively the mean and variance of $\mu_{post} - \mu_{pre}$.

Sum of Variances

Equation 10 shows the sum of variances as a comparison test. The two accuracies are different when the distance it is higher than the square root of the prequential and postquential variances.

$$\mu_{post} - \mu_{pre} > \sqrt{\sigma^2_{post} + \sigma^2_{pre}} \quad (10)$$

T-test

Equation 11 describes the t-test used to compare the prequential and postquential accuracies. We apply a one-sample t-test where we check if μ is different from 0 with a 99% confidence [101].

$$\sqrt{n}\frac{\mu}{\sigma} > 2.326 \tag{11}$$

Z-test

Equation 12 shows how we computed the two proportion z-test pooled for μ_{pre} equal μ_{post}. We first compute the z_{score}, then we compare it with the Z-value that ensures confidence of 99% [100, 111]. The z_{score} is the observed difference (a) divided by the standard error of the difference (b) pooled from the two samples (p).

$$\begin{aligned} a &= \mu_{pre} - \mu_{post} \\ p &= \frac{\mu_{pre} + \mu_{post}}{2} \\ b &= \sqrt{\frac{2p(1-p)}{n}} \\ z_{score} &= \frac{a}{b} \\ z_{score} &> 2.576 \end{aligned} \tag{12}$$

Sum of Standard Deviations

Equation 13 shows the use of the sum of standard deviations as comparison test. The difference between postquential and prequential is significant when that difference is higher than the sum of standard deviations.

$$\mu_{post} - \mu_{pre} > \sigma_{post} + \sigma_{pre} \tag{13}$$

6.2.5 Pre- and Postquential Statistics Computation

In Algorithm 4 we mention that the accuracy and its variance are evaluated both prequentially and postquentially. However, only the most recent data points are relevant for these statistics. We compared two ways of computing the mean (μ) and variance (σ^2): sliding and fading.

The sliding version uses a sliding window to store the statistics. Let $P_i \in \{0, 1\}$ be the correctness of the prediction for data point i. The values of P_i are stored in a binary sliding window W of size W_{size}. W is updated with the most recent P_i and the mean and variance of the accuracy are computed as follows:

$$\begin{aligned} \mu &= \frac{\sum_{P_i \in W} P_i}{W_{size}} \\ \sigma^2 &= \mu(1 - \mu) \end{aligned} \tag{14}$$

This expression of σ^2 comes from the fact that P_i is a binary variable.

The sliding version increases memory consumption because it needs to keep W in memory. The fading version addresses this downside of the sliding version. To reduce memory usage, the fading version relies on a fading factor [43]. The sum maintained to compute the accuracy and the variance are faded. Which mean these sums are multiplied by a fading factor $f \in [0, 1]$ before being updated. It gives a weight to all elements with older elements having a smaller weight. If $f = 1$ then the sum is computed for the entire stream. If $f = 0$ then only the last point is taken into account.

To compute the fading statistics, we need the count of data points n, the faded count of data points $N = \sum_{i=1}^{n} f^{n-i}$, and the faded accuracy of the prediction $A_n = \sum_{i=1}^{n} f^{n-i} P_i$. This is sufficient to compute the mean accuracy and its variance:

$$\begin{aligned} \mu &= \frac{A_n}{N} \\ \sigma^2 &= \mu(1 - \mu) \end{aligned} \tag{15}$$

In this experiment, we use a fading factor of 0.995.

6.2.6 Tree Addition Method

In the Data Stream Mondrian forest, adding a tree simply implants a root in the node pool. The main issue in adding trees revolves around the number of nodes available for that tree to grow. Indeed, if the number of nodes available is too small, the tree won't grow much and it will underfit the data.

Therefore, to make space for new trees, we need to trim the leaves of existing trees. The trimming phase ensures that every tree has a similar size while accommodating enough memory space for the new tree to grow. We test three approaches for trimming trees: Add random, Add depth, and Add count. The Add random approach randomly selects the leaves to remove. The Add depth approach removes the deepest leaves first. The Add count approach focuses on the leaves that contain the least amount of data points.

6.2.7 Datasets

We used six datasets to evaluate our proposed methods: three synthetic datasets to mimic real-world situations and to make comparisons with and without concept drifts, and two real Human Activity Recognition datasets.

Banos *et al*

The Banos *et al* dataset [9][1] is a human activity dataset with 17 participants and 9 sensors per participant. Each sensor samples a 3D acceleration, gyroscope, and magnetic field, as well as the orientation in a quaternion format, producing a total of 13 values. Sensors are sampled at 50 Hz, and each sample is associated with one of 33 activities. In addition to the 33 activities, an extra activity labeled 0 indicates no specific activity.

We pre-processed the Banos *et al* dataset as in [13], using non-overlapping windows of one second (50 samples), and using only the 6 axes (acceleration and gyroscope) of the right forearm sensor. We computed the average and standard deviation over the window as features for each axis, similar to the second feature set in [13]. Indeed, this feature set was providing the best performance while minimizing the producing cost.. We assigned the most frequent label to the window. The resulting data points were shuffled uniformly.

In addition, we constructed another dataset from Banos *et al*, in which we simulated a concept drift by shifting the activity labels in the second half of the data stream.

Recofit

The Recofit dataset [79, 80] is a human activity dataset containing 94 participants. Similar to the Banos *et al* dataset, the activity labeled 0 indicates no specific activity. Since many of these activities were similar, we merged some of them together using the same logic as in Table 4.1.

We pre-processed the dataset similarly to the Banos *et al* dataset, using non-overlapping windows of one second, and only using 6 axes (acceleration and gyroscope) from one sensor. From these 6 axes, we used the average and the standard deviation over the window as features. We assigned the most frequent label to the window.

[1] available here

MOA Datasets

We generated two synthetic datasets using Massive Online Analysis [18] (MOA) is a Java framework to compare data stream classifiers. In addition to classification algorithms, MOA provides many tools to read and generate datasets. We generate two synthetic datasets (MOA commands available here) using the RandomRBF algorithm, a stable dataset, and a dataset with a drift. Both datasets have 12 features and 33 labels, similar to the Banos *et al* dataset. We generated 20,000 data points for each of these synthetic datasets.

Covtype

The Covtype dataset[2] is a tree dataset. Each data point is a tree described by 54 features including ten quantitative variables and 44 binary variables. The 581,012 data points are labeled with one of the seven forest cover types and these labels are highly imbalanced. In particular, two labels represent 85% of the dataset.

6.2.8 Evaluation Metric

The models start without prior knowledge of the datasets. We evaluated our methods using a prequential fading macro F1-score. We focused on the F1 score because most datasets are imbalanced. We used the prequential evaluation because we process a data stream [43]. The prequential evaluation or *interleaved-test-then-train* evaluation is the most popular method to evaluate data stream models. It first tests the model with the data points, then trains the model with it. We used a fading factor to minimize the impact of old data points, especially data points at the beginning or data points seen before a drift occur. To obtain this fading F1 score, we multiplied the confusion matrix with the fading factor before incrementing the cell in the confusion matrix. The model is continuously evaluated throughout the data stream and we report only the final evaluation metric. The F1 scores are averaged across 20 repetitions.

6.3 Results

In this section, we first highlight the existence of an optimal number of trees dependent on the data and the memory. Then we evaluate the performance of the tree-adding methods

[2]available here

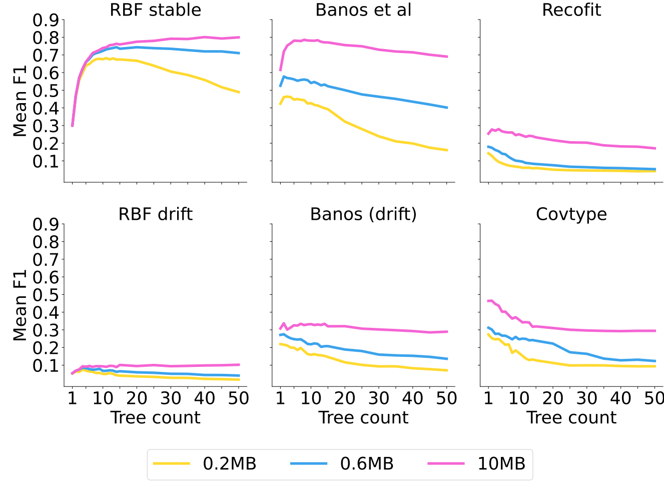


Figure 6.1: The impact of the ensemble size on the F1 score depending on the datasets and the memory limit.

independently from the dynamic update process. Finally, we assess the complete dynamic update method and all its parameter.

6.3.1 Optimal Forest Size

Figure 6.1 shows the relation between the number of trees in the Mondrian forest and the F1 score. We notice that in most configurations, there is an optimal number of trees located between 1 and 15, except for 10MB and the datasets RBF stable, in which case the F1 score keeps increasing without reaching a maximum.

A particular situation occurs on the Covtype and the Recofit datasets with 0.2MB and 0.6MB: the optimal ensemble size is one, which suggests that the memory limit is not high enough for a tree to overfit the datasets. Therefore, adding trees will always underfit.

We observe significant performance differences between the best-performing and least-performing number of trees, in particular for low memory amounts. Therefore, optimizing the number of trees is necessary to achieve the best performances.

6.3.2 Tree Addition

Figure 6.2 compares the tree addition methods presented in Section 6.2.6 with Fixed, the Mondrian forest with a fixed number of trees described in Section 6.3.1. This comparison is done independently of the dynamic procedure that will be evaluated later. In this Figure, Add random, Add count, and Add depth, start their forest with 1 tree, then add a tree

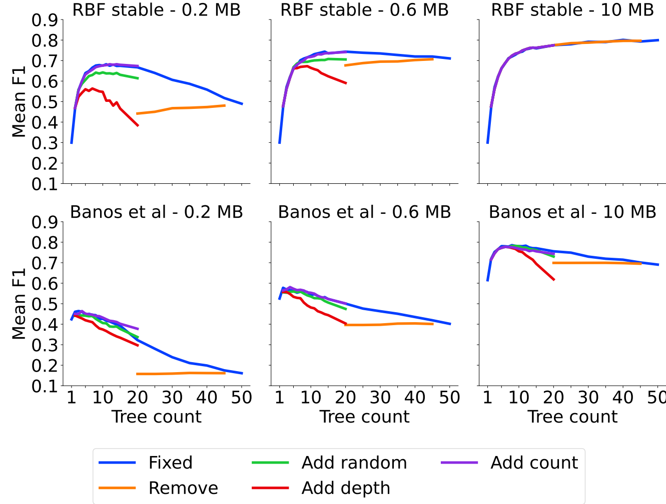


Figure 6.2: The effectiveness of the adding methods and removing method compare to Fixed.

periodically until the forest reaches the size given on the x-axis. Ideally, the update method should be indistinguishable from Fixed that uses the number of trees indicated on the x-axis.

From Figure 6.2, we note that Add Count is consistently closer to Fixed than Add random is. Add depth tend to be the worst variant to add a new tree as it diverges faster than the other two. Therefore, adding trees with Add count is a functionnal strategy to dynamicly adjust the number of tree.

6.3.3 Tree Removal

Since detecting underfitting is challenging, tree removal is not included in Algorithm 4. However, we implemented a tree removal method that we evaluated similarly to the add method, in Figure 6.2. The Remove method in that Figure starts a forest with 50 trees and periodically removes a tree to reach the value indicated on the x-axis.

We note that removing trees always underperforms except for the highest amounts of memory. Indeed, removing a tree to decrease underfitting by allowing the remaining trees to grow more nodes raises an issue related to outlier data points. The forest deletes a tree when the memory limit has been reached. Therefore, the tree growth has been paused and only the node statistics (box and counters) are updated. Once a tree is deleted, the other trees resume their growth until the memory is full again. However, during the pause phase, data points have still been received, but since none of the points outside a box could branch off, they forced the boxes to expand so they fit all data points. Thus, the node boxes tend to be bigger.

When growth is resumed, data points that could introduce new nodes are more likely to be outliers since only data points outside a box can branch off. Therefore, most of the new nodes, if not all, will be created for outliers, and these nodes are unlikely to be used and to improve the classification.

6.3.4 Comparison to Fixed Ensemble Size

We tested all possible combinations of:

- Tree addition methods (Section 6.2.6)
- Pre- and Postquential statistics (Section 6.2.5)
- Comparison Tests (Section 6.2.4)

The tree addition includes Add random, Add depth, and Add count. The pre- and postquential statistics include fading and sliding. Finally, the comparison test contains the sum of standard deviation (sum-std), the sum of variance (sum-var), the t-test (t-test), and the z-test (z-test).

Figure 6.3 shows how the top combinations compare to Fixed for each dataset and memory limit. The score is relative to the F1 score of Fixed with the optimal number of trees.

We observe that for most datasets, at least one dynamic forest reaches the performance of the Fixed method. The only exception is the RBF drift dataset with 0.2MB where all the dynamic approaches are substantially under the performance obtained by the Fixed method.

For the drift datasets, some dynamic forests surpass the performance of the Fixed method. This is due to the introduction of new trees that are not influenced by older concepts, and thus are more accurate to the new data points.

Nevertheless, no single dynamic method consistently reaches the performance of Fixed. Indeed, the count fading t-test method (purple on Figure 6.3) reaches or surpasses the performance of the Fixed method for RBF stable and drift (except 0.2MB), Banos *et al* and Banos *et al* drift, however, it underperforms on the Recofit and Covtype datasets where the count fading sum-std (blue on Figure 6.3) and the random fading sum-std (orange on Figure 6.3) perform significantly better. Conversely, the depth fading sum-std method (brown on Figure 6.3) reaches the Fixed performance for Covtype, Recofit, and Banos *et al* datasets, but significantly underperforms on RBF stable and RBF drift.

We computed the average rank of the dynamic forests and reported the top 10 methods in Table 6.1. The best average rank of 6.50 out of 24 indicates the lack of a clear winner among all combinations.

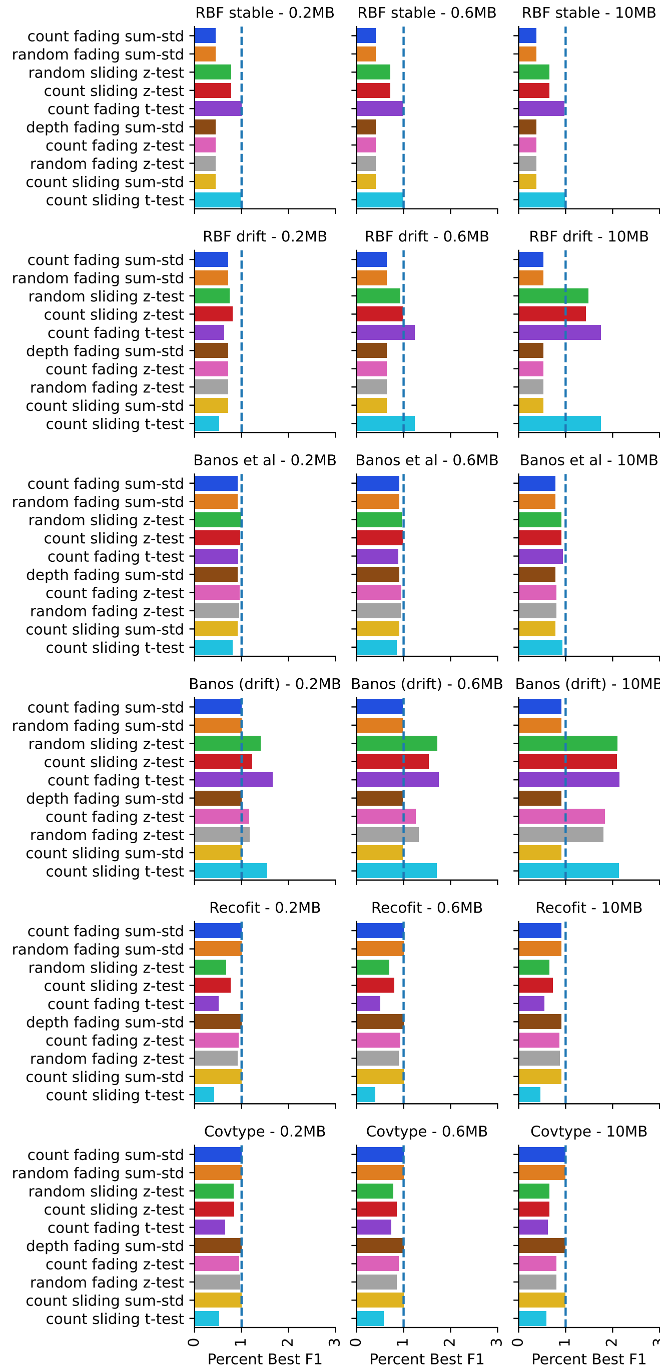


Figure 6.3: Comparison between component combinations and Fixed with the optimal tree count. Fixed is represented by a vertical dashed line. The value for the component combinations is percentage of the Fixed F1 score.

Moreover, we observed that varying the fading factor made some methods consistently approach the Fixed performance. In particular, when reducing the fading factor, the methods

Tree Addition	Pre & Postquential	Comp. Test	Avg. Rank
count	fading	sum-std	6.50
random	fading	sum-std	7.50
random	sliding	z-test	7.61
count	sliding	z-test	7.67
count	fading	t-test	8.39
depth	fading	sum-std	8.50
count	fading	z-test	8.67
random	fading	z-test	9.28
count	sliding	sum-std	9.50
count	sliding	t-test	10.44

Table 6.1: The best-ranked component combinations out of the 24 combinations of Algorithm 4 across datasets and memory limits. The top lines rank better on average than the bottom lines.

using the z-test and the t-test approach Fixed on all datasets. This is explained by the fact that the experimental conditions are closer to the assumptions made by the t-test and the z-test. Nevertheless, these fading factor explorations were not done with a proper cross-validation, and cannot be repported in this paper because it would require an analysis on independent datasets.

6.4 Related Work

The work in [85, 36] propose methods to adjust ensemble size based on accuracy contribution of the sub-classifier. A sub-classifier that significantly decreases the accuracy of the ensemble should be removed, whereas the new sub-classifier built on the last chunk of data should be added if it significantly increases the ensemble accuracy. However, their hypotheses are not valid in our study, since growing a new tree influences the performance of existing trees. Indeed, the new tree requires nodes to grow and these nodes will be taken from existing trees.

The work in [75] explores an algorithm to adjust hyper-parameters on data streams with concept drift. When a concept drift is detected, the algorithm makes a list of hyper-parameters configurations to evaluate on the most recent data points. The best configuration provides the new hyper-parameters until the next concept drift. This method is not suited

for our hypothesis since it assumes we have enough memory to store at least two models.

6.5 Conclusion

In this paper, we showed experimentally that a memory-bounded Mondrian forest has an optimal ensemble size that depends on the dataset and the memory limit. To find this optimal, we proposed a dynamic ensemble-sized Mondrian forest that estimates overfitting to drive the ensemble toward the optimal number of trees. The overfitting measure relies on the postquential accuracy, an innovative concept to estimate the training accuracy of a data stream classifier.

We introduced the use of fading factor to keep track of the mean and variance, needed for the comparison test. We tested our algorithm on six datasets with different combinations of comparison tests, different methods to add trees, and different ways of computing the mean and variance.

From this experiment, we observed that some of the proposed methods were able to reach the performance of a Mondrian forest with an optimal number of trees. However, none of the methods consistently achieve that optimal. In future work, we suggest investigating the role of the fading factor.

In addition, further investigations are required to design a functional tree removal method that resists two issues: detecting underfitting in the forest and continuing the growth of paused Mondrian trees.

Chapter 7

Conclusion

7.1 Benchmark Contributions

This manuscript started with an evaluation of several data stream classification models, focusing on their resource usage and their classification performance. Given the importance of memory usage for connected objects, we noted the current lack of memory-constrained data stream classification models in the literature. Our evaluation revealed the top three classifiers to be the Hoeffding tree, the Naïve Bayes, and the Mondrian forest, although their classification performance are significantly below the one obtained with offline kNN. Interestingly, our results also showed that the Mondrian forest's classification performance is sensitive to the amount of available memory, and with sufficient memory, the model can achieve state-of-the-art performance. As a result, we chose to focus this thesis on exploring the topic of memory-constrained Mondrian forests for data stream classification.

We considered using deep learning algorithms but they are generally unsuited for data stream classification. Indeed, it often requires processing the dataset multiple times, which is not feasible for connected devices. Even though pre-trained feature extraction models could provide better features, such models often need significant computation resources that may not be available on these devices. However, these model offers the advantage of a fixed memory footprint.

7.2 Mondrian Forest Contributions

We have developed and evaluated five mechanisms to enable the Mondrian forest to operate efficiently under memory constraints. These mechanisms define how the Mondrian tree

updates its node's boxes and counters when the memory is full. In addition, we introduced and evaluated pruning methods that enable the Mondrian forest to handle concept drift and optimize the tree over time. Our updating mechanisms significantly improved the Mondrian forest over the original algorithm described in [71], allowing it to operate efficiently in memory-constrained environments. The pruning methods also improved classification performance on data streams with concept drift. Moreover, our experiments revealed an interesting trade-off between tree depth and the number of trees in a memory-constrained Mondrian forest. We found that there is an optimal size for the Mondrian forest beyond which its performance declines. This trade-off arises because the model must choose between tree depth and the number of trees when operating under memory constraints.

Since this optimal ensemble size depends on various factors such as the dataset or the memory limit, we developed and evaluated a method to extend the ensemble toward that optimal number of trees. This method measures the overfitting of the ensemble, and when it is significantly high, a new tree is introduced in the forest. Starting with only one tree in the forest, our results demonstrate that the Mondrian forest reaches the optimal size as well as optimal classification performance in most datasets. Furthermore, in the event of a concept drift, the method surpasses the optimal performance by introducing more up-to-date trees.

7.3 Other Contributions

All the methods discussed in this manuscript are available in OrpailleCC, which is an open-source collection of data stream algorithms. We implemented these algorithms with deployment on embedded devices in mind. Therefore, our implementation does not assume the availability of kernel functions like *malloc*. Depending on the algorithm used, either the memory will be reserved automatically by the implementation or the user must provide a kernel function to allocate memory. This flexibility allows the user to adapt the implementation to the specific constraints and requirements of their system.

The algorithmic approach followed in this manuscript is not the only way to address memory-constrained models. We collaborated on a numerical stability study of the Mondrian forest that showed the numerical precision can be reduced to 8-bit without impacting the classification performance [107]. This study used Verificarlo [32] to replace 64-bit floating point operations with their 8-bit equivalent. This approach aims at reducing the memory footprint of the model through the floating-point format and can be applied to other models.

Overall, this thesis contributes to improving the classification performance of the Mondrian forest for memory-constrained environments, with a focus on limited memory resources ranging from 200 KB to 1 MB. In these scenarios, the Mondrian forest proves to be an efficient solution for classification tasks, particularly when it include all our extensions. However, the classification performance may vary depending on the dataset's characteristics, such as the number of labels and features, and the Hoeffding Tree or the Naïve Bayes may exibit competitive performances.

7.4 Future Work

As future work, there are several ways to further improve the Mondrian forest. For instance, developing a tree removal method that preserves the Mondrian forest's classification performances. This method would improve the ensemble size adjustment by correcting for errors in the adjustment or for concept drifts. We could also optimize the pruning method so the pruning rates adapt to the ensemble size and the memory limit.

During our research, we found that memory-constrained classification models have not been studied extensively. While the thesis focused on the Mondrian forest, there are several other models that could benefit from investigating how a memory constraint would impact them. In particular, ensemble-based models are likely to face similar issues as the Mondrian forest, such as the trade-off between the ensemble size and classification performance, and could benefit from our ensemble size adjusting algorithm. Therefore, further research could investigate the effects of memory constraints on other ensemble-based models to better understand their performance limitations in memory-constrained environments. Additionnaly, forest-based model may benefit from our trimming mechanisms.

Bibliography

[1] Hanady Abdulsalam, David B. Skillicorn, and Patrick Martin. Classification Using Streaming Random Forests. *IEEE Transactions on Knowledge and Data Engineering*, 23(1):22–36, jan 2011.

[2] Charu C Aggarwal. On biased reservoir sampling in the presence of stream evolution. In *Proceedings of the 32nd international conference on Very large data bases*, pages 607–618. VLDB Endowment, 2006.

[3] Nadeem Ahmed, Raihan Kabir, Airin Rahman, Al Momin, and Md Rashedul Islam. Smartphone sensor based physical activity identification by using hardware-efficient support vector machines for multiclass classification. In *2019 IEEE Eurasia Conference on IOT, Communication and Engineering (ECICE)*, pages 224–227. IEEE, 2019.

[4] Ian F Akyildiz, Weilian Su, Yogesh Sankarasubramaniam, and Erdal Cayirci. Wireless Sensor Networks: a Survey. *Computer Networks*, 38(4):393–422, 2002.

[5] Albert Bifet and Ricard Gavaldà. Learning from Time-Changing Data with Adaptive Windowing. In *Proceedings of the 2007 SIAM International Conference on Data Mining*, pages 443–448, Minneapolis, United States, April 2007. Society for Industrial and Applied Mathematics.

[6] N. S. Altman. An introduction to kernel and nearest-neighbor nonparametric regression. *The American Statistician*, 46(3):175, aug 1992.

[7] Giuseppe Amato, Davide Bacciu, Stefano Chessa, Mauro Dragone, Claudio Gallicchio, Claudio Gennaro, Hector Lozano, Alessio Micheli, Gregory O'Hare, Arantxa Renteria, and Claudio Vairo. A Benchmark Dataset for Human Activity Recognition and Ambient Assisted Living. *In International Symposium on Ambient Intelligence*, pages 1–9, 01 2016.

[8] Amazon Web Services. FreeRTOS. https://www.freertos.org/. Accessed: 2023-01-23.

[9] Oresti Baños, Miguel Damas, Héctor Pomares, Ignacio Rojas, Máté Attila Tóth, and Oliver Amft. A Benchmark Dataset to Evaluate Sensor Displacement in Activity Recognition. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, UbiComp '12, page 1026–1035, New York, NY, USA, 2012. Association for Computing Machinery.

[10] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and Issues in Data Stream Systems. In *Proceedings of the Twenty-First ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '02, page 1–16, New York, NY, USA, 2002. Association for Computing Machinery.

[11] Brian Babcock, Mayur Datar, and Rajeev Motwani. Sampling from a moving window over streaming data. In *Proceedings of the thirteenth annual Association for Computing Machinery-SIAM symposium on Discrete algorithms*, pages 633–634. Society for Industrial and Applied Mathematics, 2002.

[12] Balaji Lakshminarayanan. Python implementation of the mondrian forest, 2014.

[13] Oresti Banos, Juan-Manuel Galvez, Miguel Damas, Hector Pomares, and Ignacio Rojas. Window Size Impact in Human Activity Recognition. 14(4):6474–6499, April 2014.

[14] András A. Benczúr, Levente Kocsis, and Róbert Pálovics. *Online Machine Learning Algorithms over Data Streams*, pages 1199–1207. Springer International Publishing, Cham, 2019.

[15] Gérard Biau and Erwan Scornet. A random forest guided tour. *TEST*, 25(2):197–227, apr 2016.

[16] Albert Bifet and Ricard Gavaldà. Adaptive Learning from Evolving Data Streams. In *Advances in Intelligent Data Analysis VIII*, pages 249–260, Lyon, France, August 2009. Springer Berlin Heidelberg.

[17] Albert Bifet and Ricard Gavaldà. Adaptive learning from evolving data streams. In *Advances in Intelligent Data Analysis VIII*, pages 249–260. Springer Berlin Heidelberg, 2009.

[18] Albert Bifet, Geoff Holmes, Richard Kirkby, and Bernhard Pfahringer. MOA: Massive Online Analysis. *Journal of Machine Learning Research*, 11(May):1601–1604, 2010.

[19] Albert Bifet, Geoff Holmes, Bernhard Pfahringer, Richard Kirkby, and Ricard Gavaldà. New Ensemble Methods for Evolving Data Streams. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '09*, Paris, France, June 2009. Association for Computing Machinery.

[20] Albert Bifet, Silviu Maniu, Jianfeng Qian, Guangjian Tian, Cheng He, and Wei Fan. StreamDM: Advanced Data Mining in Spark Streaming. In *Proceedings of the 2015 IEEE International Conference on Data Mining Workshop*, ICDMW '15, page 1608–1611, USA, 2015. IEEE Computer Society.

[21] Albert Bifet, Jiajin Zhang, Wei Fan, Cheng He, Jianfeng Zhang, Jianfeng Qian, Geoff Holmes, and Bernhard Pfahringer. Extremely Fast Decision Tree Mining for Evolving Data Streams. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '17*, pages 1733–1742. Association for Computing Machinery, August 2017.

[22] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, jul 1970.

[23] Leo Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, aug 1996.

[24] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[25] Gilles Callebaut, Guus Leenders, Jarne Van Mulders, Geoffrey Ottoy, Lieven De Strycker, and Liesbet Van der Perre. The Art of Designing Remote IoT Devices—Technologies and Strategies for a Long Battery Life. *Sensors*, 21(3):913, 2021.

[26] Alberto Cano and Bartosz Krawczyk. Kappa Updated Ensemble for Drifting Data Stream Mining. *Machine Learning*, 109(1):175–218, 2020.

[27] Poornima M Chanal and Mahabaleshwar S Kakkasageri. Security and privacy in iot: a survey. *Wireless Personal Communications*, 115:1667–1693, 2020.

[28] Ricardo Chavarriaga, Hesam Sagha, Alberto Calatroni, Sundara Tejaswi Digumarti, Jose del R. Millan, Daniel Roggen, and Gerhard Tröster. The Opportunity challenge: A benchmark database for on-body sensor-based activity recognition. *Pattern Recognition Letters*, 23:2033–2042, 01 2013.

[29] Sakshi Chhabra and Dinesh Singh. Data Fusion and Data Aggregation/Summarization Techniques in WSNs: A Review. *International Journal of Computer Applications*, 121(19), 2015.

[30] A. P. Dawid. Present position and potential developments: Some personal views: Statistical theory: The prequential approach. *Journal of the Royal Statistical Society. Series A (General)*, 147(2):278, 1984.

[31] Akbar Dehghani, Omid Sarbishei, Tristan Glatard, and Emad Shihab. A Quantitative Comparison of Overlapping and Non-Overlapping Sliding Windows for Human Activity Recognition Using Inertial Sensors. *Sensors*, 19(22):5026, 2019.

[32] Christophe Denis, Pablo De Oliveira Castro, and Eric Petit. Verificarlo: Checking Floating Point Accuracy through Monte Carlo Arithmetic. In *2016 IEEE 23nd Symposium on Computer Arithmetic (ARITH)*, pages 55–62, 2016.

[33] Pedro Domingos and Geoff Hulten. Mining High-Speed Data Streams. *Proceeding of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 11 2002.

[34] Xibin Dong, Zhiwen Yu, Wenming Cao, Yifan Shi, and Qianli Ma. A Survey on Ensemble Learning. *Frontiers of Computer Science*, 14:241–258, 2020.

[35] E.J. Duarte-Melo and Mingyan Liu. Analysis of energy consumption and lifetime of heterogeneous wireless sensor networks. In *Global Telecommunications Conference. GLOBECOM 02*. IEEE, 2002.

[36] Piotr Duda, Maciej Jaworski, and Leszek Rutkowski. On ensemble components selection in data streams scenario with reoccurring concept-drift. In *2017 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1–7, 2017.

[37] Dr. Lachit Dutta and Swapna Bharali. TinyML Meets IoT: A Comprehensive Survey. *Internet of Things*, 16:100461, 2021.

[38] Sanem Elbasi, Alican Büyükçakır, Hamed Bonab, and Fazli Can. On-the-Fly Ensemble Pruning in Evolving Data Streams, 2021.

[39] Bin Fan, Dave G. Andersen, Michael Kaminsky, and Michael D. Mitzenmacher. Cuckoo filter. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies - CoNEXT '14*. ACM Press, 2014.

[40] Bin Fan, Dave G. Andersen, Michael Kaminsky, and Michael D. Mitzenmacher. Cuckoo Filter: Practically Better Than Bloom. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14, page 75–88, New York, NY, USA, 2014. Association for Computing Machinery.

[41] Jessica Fernandes, Everton Santana, Victor Turrisi da Costa, Bruno Bogaz Zarpelão, and Sylvio Barbon. Evaluating the Four-way Performance Trade-off for Data Stream Classification in Edge Computing. *IEEE Transactions on Network and Service Management*, PP:1–1, mar 2020.

[42] Yoav Freund and Robert E Schapire. A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting. *Journal of Computer and System Sciences*, 55(1):119–139, aug 1997.

[43] João Gama, Raquel Sebastião, and Pedro Pereira Rodrigues. Issues in Evaluation of Stream Learning Algorithms. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '09, page 329–338, Paris, France, 2009. Association for Computing Machinery.

[44] João Gama, Pedro Medas, and Ricardo Rocha. Forest trees for on-line data. 2004.

[45] Pierre Geurts, Damien Ernst, and Louis Wehenkel. Extremely randomized trees. *Machine learning*, 63(1):3–42, 2006.

[46] Corrado Gini. Concentration and dependency ratios. *Rivista di politica economica*, 87:769–792, 1997.

[47] Lukasz Golab and M. Tamer Özsu. Issues in Data Stream Management. *SIGMOD Rec.*, 32(2):5–14, jun 2003.

[48] Heitor M. Gomes, Albert Bifet, Jesse Read, Jean Paul Barddal, Fabrício Enembreck, Bernhard Pfharinger, Geoff Holmes, and Talel Abdessalem. Adaptive Random Forests for Evolving Data Stream Classification. *Machine Learning*, 106(9-10):1469–1495, June 2017.

[49] Heitor Murilo Gomes, Jesse Read, Albert Bifet, Jean Paul Barddal, and João Gama. Machine Learning for Streaming Data: State of the Art, Challenges, and Opportunities. *SIGKDD Explor. Newsl.*, 21(2):6–22, November 2019.

[50] Maciej Grzenda, Heitor Murilo Gomes, and Albert Bifet. Delayed Labelling Evaluation for Data Streams. *Data Mining and Knowledge Discovery*, 34(5):1237–1266, 2020.

[51] Chirag Gupta, Arun Sai Suggala, Ankit Goyal, Harsha Vardhan Simhadri, Bhargavi Paranjape, Ashish Kumar, Saurabh Goyal, Raghavendra Udupa, Manik Varma, and Prateek Jain. ProtoNN: Compressed and Accurate kNN for Resource-scarce Devices. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 1331–1340, Sydney NSW Australia, August 2017. Proceedings of Machine Learning Research.

[52] Peter J Haas. Data-Stream Sampling: Basic Techniques and Results. *Data Stream Management: Processing High-Speed Data Streams*, pages 13–44, 2016.

[53] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA data mining software. *ACM SIGKDD Explorations Newsletter*, 11(1):10, nov 2009.

[54] HelloAlone. C++ implementation of the mondrian forest, 2018.

[55] Nagaraj Honnikoll and Ishwar Baidari. Mean Error Rate Weighted Online Boosting Method. *The Computer Journal*, October 2021.

[56] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An introduction to statistical learning*, volume 112. Springer, 2013.

[57] Majid Janidarmian, Atena Roshan Fekr, Katarzyna Radecka, and Zeljko Zilic. A Comprehensive Analysis on Wearable Acceleration Sensors in Human Activity Recognition. *Sensors*, 17(3):529, mar 2017.

[58] Amrinder Kaur and Rakesh Kumar. A Comprehensive Analysis of Classification Methods for Big Data Stream. In Harish Sharma, Kannan Govindan, Ramesh C. Poonia, Sandeep Kumar, and Wael M. El-Medany, editors, *Advances in Computing and Intelligent Systems*, pages 213–222, Singapore, 2020. Springer Singapore.

[59] Arun Kejariwal, Sanjeev Kulkarni, and Karthik Ramasamy. Real time analytics: : algorithms and systems. *Proceedings of the VLDB Endowment*, 8(12):2040–2041, aug 2015.

[60] Martin Khannouz and Tristan Glatard. A Benchmark of Data Stream Classification for Human Activity Recognition on Connected Objects. *Sensors*, 20(22):6486, 2020.

[61] Martin Khannouz and Tristan Glatard. Dynamic Ensemble Size Adjustment for Memory Constrained Mondrian Forest. In *2022 IEEE International Conference on Big Data (Big Data)*, pages 3358–3363, Osaka, Japan, dec 2022. IEEE Computer Society.

[62] Martin Khannouz and Tristan Glatard. Mondrian Forest for Data Stream Classification Under Memory Constraints, 2022.

[63] Martin Khannouz, Bo Li, and Tristan Glatard. OrpailleCC: a Library for Data Stream Analysis on Embedded Systems. *Journal of Open Source Software*, 4(39):1485, 2019.

[64] Taghi M. Khoshgoftaar, Moiz Golawala, and Jason Van Hulse. An Empirical Study of Learning from Imbalanced Data Using Random Forest. In *19th IEEE International Conference on Tools with Artificial Intelligence(ICTAI 2007)*. IEEE, oct 2007.

[65] J. Kittler. Combining classifiers: A theoretical framework. *Pattern Analysis and Applications*, 1(1):18–27, mar 1998.

[66] Ron Kohavi and Clayton Kunz. Option decision trees with majority votes. In *ICML*, volume 97, pages 161–169, 1997.

[67] Sotiris B Kotsiantis, I Zaharakis, and P Pintelas. Supervised machine learning: A review of classification techniques. *Emerging artificial intelligence applications in computer engineering*, 160:3–24, 2007.

[68] Helena Kotthaus, Ingo Korb, Michel Lang, Bernd Bischl, Jörg Rahnenführer, and Peter Marwedel. Runtime and memory consumption analyses for machine learning R programs. *Journal of Statistical Computation and Simulation*, 85(1):14–29, jan 2015.

[69] Vrushali Y Kulkarni and Pradeep K Sinha. Pruning of random forest classifiers: A survey and future directions. jul 2012.

[70] Ashish Kumar, Saurabh Goyal, and Manik Varma. Resource-Efficient Machine Learning in 2 KB RAM for the Internet of Things. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, page 1935–1944, Sydney, NSW, Australia, August 2017. JMLR.org.

[71] Balaji Lakshminarayanan, Daniel M Roy, and Yee Whye Teh. Mondrian Forests: Efficient Online Random Forests. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, volume 4, pages 3140–3148. Curran Associates, Inc., Montreal, Canada, December 2014.

[72] David D. Lewis. Naive (Bayes) at Forty: The Independence Assumption in Information Retrieval. In *Proceedings of the 10th European Conference on Machine Learning*, ECML '98, pages 4–15, Berlin, Heidelberg, 1998. Springer-Verlag.

[73] Bo Li, Omid Sarbishei, Hosein Nourani, and Tristan Glatard. A multi-dimensional extension of the Lightweight Temporal Compression method. In *2018 IEEE International Conference on Big Data*, pages 2918–2923. IEEE, dec 2018.

[74] Bo Li, Omid Sarbishei, Hosein Nourani, and Tristan Glatard. A multi-dimensional extension of the Lightweight Temporal Compression method. In *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, dec 2018.

[75] Jesus L. Lobo, Javier Del Ser, and Eneko Osaba. Lightweight Alternatives for Hyperparameter Tuning in Drifting Data Streams. In *2021 International Conference on Data Mining Workshops (ICDMW)*, pages 304–311, 2021.

[76] Wei-Yin Loh and Yu-Shan Shih. Split selection methods for classification trees. *Statistica sinica*, pages 815–840, 1997.

[77] Sam Lucero et al. IoT platforms: enabling the Internet of Things. *White paper*, 2016.

[78] Jacob Montiel, Albert Bifet, Viktor Losing, Jesse Read, and Talel Abdessalem. Learning Fast and Slow: A Unified Batch/Stream Framework. In *2018 IEEE International Conference on Big Data (Big Data)*, Seattle, WA, USA, December 2018. Institute of Electrical and Electronics Engineers.

[79] Dan Morris, T. Scott Saponas, Andrew Guillory, and Ilya Kelner. Recofit: Using a Wearable Sensor to Find, Recognize, and Count Repetitive Exercises. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '14, page 3225–3234, Toronto, Ontario, Canada, April 2014. Association for Computing Machinery.

[80] Dan Morris, T. Scott Saponas, Andrew Guillory, and Ilya Kelner. RecoFit: Using a Wearable Sensor to Find, Recognize, and Count Repetitive Exercises, 2014.

[81] M. G. Sarwar Murshed, Christopher Murphy, Daqing Hou, Nazar Khan, Ganesh Ananthanarayanan, and Faraz Hussain. Machine Learning at the Network Edge: A Survey. *ACM Comput. Surv.*, 54(8), October 2021.

[82] Mohamed A. Nassar, Len Luxford, Peter Cole, Giles Oatley, and Polychronis Koutsakis. Adaptive Low-Power Wireless Sensor Network Architecture for Smart Street Furniture-based Crowd and Environmental Measurements. In *2019 IEEE 20th International Symposium on "A World of Wireless, Mobile and Multimedia Networks" (WoWMoM)*, pages 1–9, 2019.

[83] Hai-Long Nguyen, Yew-Kwong Woon, and Wee-Keong Ng. A Survey on Data Stream Clustering and Classification. *Knowledge and information systems*, 45:535–569, 2015.

[84] N.C. Oza. Online Bagging and Boosting.

[85] Lena Pietruczuk, Leszek Rutkowski, Maciej Jaworski, and Piotr Duda. A Method for Automatic Adjustment of Ensemble Size in Stream Data Mining. In *2016 International Joint Conference on Neural Networks (IJCNN)*, pages 9–15, 2016.

[86] Bakshi Prasad and Sonali Agarwal. Stream Data Mining: Platforms, Algorithms, Performance Evaluators and Research Trends. *International Journal of Database Theory and Application*, 9:201–218, 09 2016.

[87] S Priya and R Annie Uthra. Comprehensive analysis for class imbalance data with concept drift using ensemble based classification. *Journal of Ambient Intelligence and Humanized Computing*, apr 2020.

[88] Junfei Qiu, Qihui Wu, Guoru Ding, Yuhua Xu, and Shuo Feng. A survey of machine learning for big data processing. *EURASIP Journal on Advances in Signal Processing*, 2016:1–16, 2016.

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

[90] Anand Rajaraman and Jeffrey David Ullman. *Mining of Massive Datasets*. Cambridge University Press, USA, 2011.

[91] Partha Pratim Ray. A Review on TinyML: State-of-the-Art and Prospects. *Journal of King Saud University - Computer and Information Sciences*, 34(4):1595–1623, April 2022.

[92] Leszek Rutkowski, Lena Pietruczuk, Piotr Duda, and Maciej Jaworski. Decision trees for mining data streams based on the mcdiarmid's bound. *IEEE Transactions On Knowledge And Data Engineering*, 2013.

[93] Amir Saffari, Christian Leistner, Jakob Santner, Martin Godec, and Horst Bischof. On-line Random Forests. In *2009 IEEE 12th International Conference on Computer Vision Workshops, ICCV Workshops*. IEEE, sep 2009.

[94] Rubn San-Segundo, Henrik Blunck, Jos Moreno-Pimentel, Allan Stisen, and Manuel Gil-Martn. Robust Human Activity Recognition Using Smartwatches and Smartphones. *Engineering Applications of Artificial Intelligence*, 72(C):190–202, June 2018.

[95] O. Sarbishei. A platform and methodology enabling real-time motion pattern recognition on low-power smart devices. In *2019 IEEE 5th World Forum on Internet of Things (WF-IoT)*, pages 269–272. IEEE, April 2019.

[96] T. Schoellhammer, B. Greenstein, E. Osterweil, M. Wimbrow, and D. Estrin. Lightweight temporal compression of microclimate datasets [wireless sensor networks]. In *29th Annual IEEE International Conference on Local Computer Networks*. IEEE (Comput. Soc.), 2004.

[97] Mohamed Seliem, Khalid Elgazzar, and Kasem Khalil. Towards Privacy Preserving IoT Environments: A Survey. *Wireless Communications and Mobile Computing*, 2018:1–15, 2018.

[98] C. E. Shannon. A Mathematical Theory of Communication. *Bell System Technical Journal*, 27(3):379–423, jul 1948.

[99] Amir Sinaeepourfard, Jordi Garcia, Xavier Masip-Bruin, and Eva Marin-Tordera. A Novel Architecture for Efficient Fog to Cloud Data Management in Smart Cities. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 2622–2623, 2017.

[100] R. C. Sprinthall. *Basic Statistical Analysis (9th ed.)*. Pearson Education, 2011.

[101] Student. The probable error of a mean. *Biometrika*, pages 1–25, 1908.

[102] Yu Sugawara, Satoshi Oyama, and Masahito Kurihara. Adaptive Rotation Forests: Decision Tree Ensembles for Sequential Learning. In *2021 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 613–618, Melbourne, Australia, October 2021. Institute of Electrical and Electronics Engineers.

[103] Mark Tennant, Frederic Stahl, Omer Rana, and João Bártolo Gomes. Scalable real-time classification of data streams with concept drift. *Future Generation Computer Systems*, 75:187–199, oct 2017.

[104] Alexey Tsymbal, Mykola Pechenizkiy, and Pádraig Cunningham. Dynamic Integration with Random Forests. In *Lecture Notes in Computer Science*, pages 801–808. Springer Berlin Heidelberg, 2006.

[105] Wallace Ugulino, Débora Cardador, Katia Vega, Eduardo Velloso, Ruy Milidiú, and Hugo Fuks. Wearable Computing: Accelerometers' Data Classification of Body Postures and Movements. In *Advances in Artificial Intelligence - SBIA 2012*, pages 52–61, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[106] Tim van Kasteren, Gwenn Englebienne, and B. Krose. Human Activity Recognition from Wireless Sensor Network Data: Benchmark and Software. *Activity recognition in pervasive intelligent environments*, 4:165–186, 05 2011.

[107] Marc Vicuna, Martin Khannouz, Gregory Kiar, Yohan Chatelain, and Tristan Glatard. Reducing Numerical Precision Preserves Classification Accuracy in Mondrian Forests. In *2021 IEEE International Conference on Big Data (Big Data)*, pages 2785–2790, 2021.

[108] Jeffrey S. Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software*, 11(1):37–57, mar 1985.

[109] Sebastian Wagner, Max Zimmermann, Eirini Ntoutsi, and Myra Spiliopoulou. Ageing-Based Multinomial Naive Bayes Classifiers Over Opinionated Data Streams. In Annalisa Appice, Pedro Pereira Rodrigues, Vítor Santos Costa, Carlos Soares, João Gama, and Alípio Jorge, editors, *Machine Learning and Knowledge Discovery in Databases*, pages 401–416, Cham, 2015. Springer International Publishing.

[110] Kapil K Wankhade, Snehlata S Dongre, and Kalpana C Jondhale. Data Stream Classification: A Review. *Iran Journal of Computer Science*, 3:239–260, 2020.

[111] Wikipedia contributors. Z-test — Wikipedia, the free encyclopedia, 2004. [Online; accessed 30-September-2022].