Improving Decision Tree and Neural Network Learning for Evolving Data-streams

by
Diego Marrón Vida

Advisors:
Eduard Ayguadé
José Ramón Herrero
Albert Bifet

DISSERTATION
Submitted in partial fulfilment of the requirements
for the degree of Doctor of Philosophy
in the Department of Computer Architecture

Universitat Politècnica de Catalunya
2019
Barcelona, Spain

Abstract

High-throughput real-time Big Data stream processing requires fast incremental algorithms that keep models consistent with most recent data. In this scenario, Hoeffding Trees are considered the state-of-the-art single classifier for processing data streams and they are widely used in ensemble combinations.

This thesis is devoted to the improvement of the performance of algorithms for machine learning/artificial intelligence on evolving data streams. In particular, we focus on improving the Hoeffding Tree classifier and its ensemble combinations, in order to reduce its resource consumption and its response time latency, achieving better throughput when processing evolving data streams.

First, this thesis presents a study on using Neural Networks (NN) as an alternative method for processing data streams. The use of random features for improving NNs training speed is proposed and important issues are highlighted about the use of NN on a data stream setup. These issues motivated this thesis to go in the direction of improving the current state-of-the-art methods: Hoeffding Trees and their ensemble combinations.

Second, this thesis proposes the Echo State Hoeffding Tree (ESHT), as an extension of the Hoeffding Tree to model time-dependencies typically present in data streams. The capabilities of the new proposed architecture on both regression and classification problems are evaluated.

Third, a new methodology to improve the Adaptive Random Forest (ARF) is developed. ARF has been introduced recently, and it is considered the state-of-the-art classifier in the MOA framework (a popular framework for processing evolving data streams). This thesis proposes the Elastic Swap Random Forest, an extension to ARF that reduces the number of base learners in the ensemble down to one third on average, while providing accuracy similar to that of the standard ARF with 100 trees.

And finally, a last contribution on a multi-threaded high performance scalable ensemble design that is highly adaptable to a variety of hardware platforms, ranging from server-class to edge computing. The proposed design achieves throughput improvements of 85x (Intel i7), 143x (Intel Xeon parsing from memory), 10x (Jetson TX1, ARM) and 23x (X-Gene2, ARM) compared to single-threaded MOA on i7. In addition, the proposal achieves 75% parallel efficiency when using 24 cores on the Intel Xeon.

Acknowledgements

This dissertation would not be possible without guidance and continuous support of my advisors, Eduard Ayguadé, José Ramón Herrrero, and Albert Bifet. Special mention to Nacho Navarro, for taking me as his PhD student, and giving full resources and freedom to pursue my research; I know how happy and proud he would be to see this work completed. You all have been great role models as researchers, mentors and friends.

I would like to thank to my colleagues at Barcelona Supercomputing Center, Toni Navarro, Miquel Vidal, Marc Jordà, Kevin Sala and Pau Farré. For your patience and the insane funny moments and crazy moments during this journey. I would like to mention my earlier colleagues at BSC, Lluis Vilanova and Javier Cabezas for their mentorship and help at the beginning of this PhD.

Last, but certainly not least, I am extremely grateful to Judit, Valeria and Hugo for your incredible patience, unconditional support and for inspiring me to pursue my dreams; half of this is your merit. I am also deeply grateful to my Parents, sister and bother in-law for your patience and support during these years, and most important for teaching me to never give up even when circumstances are not favourable. All of you always believed in me and wanted the best for me. Certainly, without you, none of this would have been possible.

Thank you.

Table of Contents

List of Tables

List of Figures

1 Introduction

Machine learning (ML) is an important branch of Artificial Intelligence (AI) that focuses on building and designing algorithms to make decisions or predictions about present or future events, based on learning from past events. These algorithms extract patterns and structures from examples of data in order to learn specific tasks without any human interaction or programming. The success of this learning process is heavily influenced by the amount, scope, and quality of input data.

Nowadays, a wide spectrum of technical and scientific areas use ML as part of its core development strategies: healthcare, manufacturing, travel, financial services, dynamic pricing, autonomous vehicles, fraud detection, fast adaptation to new user behavior, among many others. Its broad adoption in the industry is sparking new research opportunities, making ML one of the most active and prolific fields in computer science research today.

1.1 Supervised Machine Learning

This dissertation is focused on Supervised Machine Learning, a machine learning task that based on training dataset, builds a model capable of making predictions of new un-seen instances. The training dataset contains a set of labeled instances representing the specific task to learn. An instance is defined by the values of attributes (or features) that describe essential characteristics of the task. For example, a labeled instance can be a raw image, labeled with a *yes* if the image contains a cat, or a *no* otherwise.

The primary goal of a supervised algorithm is to infer a function that generalizes the task to learn. This supervised algorithm processes the training dataset in order to refine the inferred function, also called model. At the output level, a function quantifies the current model performance by examining the model's output for an instance (predicted label) and its correct answer (target label), and updates the model accordingly. This step is known as the induction process. A model produced by an inducer algorithm is called regressor if the output label belongs to the real-valued domain. When the output is a set of pre-defined (discretized) values, the model is called a classifier. Traditionally, an inducer algorithm builds the model by iterating over the dataset several times until the desired error is achieved, or after a maximum number of iterations. In this dissertation, we refer to this learning strategy as *batch learning* or *offline learning*.

Batch learning is a challenging process that can last for hours, days or weeks, depending on the dataset. During the learning process, algorithms typically need to deal with the following challenges: noise in data, unbalanced classes, or instances with missing values for some of the features among others. In the case of Big Data, when the dataset does not fit in memory, out-of-core or distributed learning is required, potentially affecting the final model accuracy.

In addition, offline learning works well for static problems that do not change over time. However, data generated by many of today's application can change as time passes presenting non-stationary distributions, a fact that most batch learners do not consider. For example, TCP/IP packet monitoring [21] or credit card fraud detection [98], are situations that requires fast adaptation and reaction to attackers' new techniques in order to keep them away. The usual approach in these situations is to process data on–the–fly in real time.

1.2 Processing Data Streams in Real Time

Recent advances in both hardware and software fuelled ubiquitous sources generating Big Data streams in a distributed way (Volume), at high speed (Velocity) with new data rapidly superseding old data (Volatility). For example, the Internet of Things (IoT) is the largest network of sensors and actuators connected by networks to computing systems. This includes sensors across a huge range of settings, for industrial process control, finance, health analytics, home automation, and autonomous cars, and often interconnected across domains, monitoring and functioning in people, objects and machines in real time. In these situations, storing data that is generated at an increasing volume and velocity for processing it offline can quickly become a bottleneck as time passes. Also, the uncertainty of when data will be superseded increases considerably the complexity of managing the stored data: once data is superseded a new model is required, and methods need to discriminate at which point data became irrelevant so it should not be used for building the new model. Thus, processing this type of data requires a different approach than the traditional batch learning setup: data should be processed as a continuous stream in real time.

Extracting knowledge from data streams in real time requires fast incremental algorithms that are able to deal with potentially infinite streams. In addition to the challenges already present in batch learning (see Section 1.1) processing data streams in real time imposes the following challenges:

- Deal with potentially infinite data streams: data is not stored, and used exactly once (no iteration over the dataset).
- Models must be ready to predict at any moment.
- Use limited resources (CPU time and memory) with a response time in the range of few seconds (high latency) or few milliseconds (low latency).
- Algorithms should be adaptive to the evolution of data distributions over time, since changes in them, can cause predictions to become less accurate with time (Concept Drift).

The preferred choice for processing data streams in a variety of applications is the use of incremental algorithms that can incorporate new information to the model without rebuilding it. In particular, incremental decision trees and their ensemble combinations are the preferred choices when processing Big Data streams, and are the focus of this dissertation.

1.3 Decision Trees and Ensembles for Mining Big Data Streams

In this dissertation, we focus on the usage of the Hoeffding Tree (HT) [30], an incremental decision tree that is able to learn very efficiently and with high accuracy from data streams. Usually, single learners such as the HT are combined in ensemble methods to improve their prediction performance.

The HT is considered the state-of-the-art single classifier for processing data streams. It is based on the idea that only examining a small subset of the data is enough for deciding how to grow the model. Deciding the exact size of each subset is a difficult task, which is tackled in the HT by using the Hoeffding probability bound. The main advantage of the HT is that with high probability the model produced will be similar to the one produced by an equivalent batch inducer. In this dissertation, we also use the Fast Incremental Model Trees with Drift Detection (FIMT-DD) [62], an incremental regressor tree that is also based on the Hoeffding probability bound. Both HT and FIMT-DD are further described in chapter 2.

Ensemble learners are the preferred method for processing data streams due to their better predictive performance over single models. Ensemble methods build a set of base models which are used in combination for obtaining the final label. Models built by an ensemble can use the same induction algorithm for building all models or use different inductors. In this dissertation, we only consider ensembles of homogeneous learners, and in particular, the Leveraging Bagging (LB) [13] and the Adaptive Random Forest (ARF) [44], two very popular methods for building ensembles of HT. Again, these two ensembles are further described in detail in chapter 2.

1.4 Neural Networks

Neural Networks (NN) are very popular nowadays, due to the large number of success stories when using Deep Learning (DL) methods in both the academic and industrial world. DL methods are even outperforming humans in complex tasks such as Image recognition [50], or playing Go [101], becoming the new state-of-the-art methods in Machine Learning.

Deep Learning aims for a better data representation at multiple layers of abstraction. In each layer, the network needs to be fine tuned. In classification, a common algorithm to fine tune the network is the Stochastic Gradient Descent (SGD) which minimizes the error at the output layer using an objective function, such as the mean square error. A gradient vector is used to back-propagate the error to previous layers. The gradient nature of the algorithm makes it suitable to be trained incrementally in batches of size one, similar to how standard incremental training is done.

Although deep NN can learn incrementally, they have so far proved to be too sensitive to their hyper-parameters and initial conditions; for this reason NN are not considered as an effective *off–the–shelf* solution to process data streams [80]. Observe that the ideal NN for data streams should have similar characteristics to the ones already mentioned in Section 1.2:

1. Work out-of-the-box.
2. Trained incrementally visiting each example exactly once.
3. Ready to be used at any moment.
4. Fast response time in the order of few milliseconds or few seconds.
5. React to concept drifting.

Recurrent Neural Networks (RNN) are a type of NN with an internal memory that allow them to capture temporal dependencies. Training a RNN is challenging and requires a large amount of time, making them not viable for real-time learning [113, 81]. In recent years, Reservoir Computing (RC) has emerged as an alternative to RNN, aiming for a simpler and faster training [78, 64]. Reservoir Computing can be seen as an standard NN that learns from what is called a ”reservoir” unit, which is responsible for capturing the temporal dependencies of the input data stream. Although conceptually simpler than most RNN, computationally cheap, and easier to implement, RC still have high sensitivity to hyper-parameter configurations (i.e. small changes to any of them affect the accuracy in a non-predictable way [75]).

1.5 Contributions of this Thesis

The main goals of this thesis are to improve the Hoeffding Tree and its ensemble combinations from both algorithmic and implementation point of view, in order to be able to provide higher throughput and to reduce the consumption of resources while providing similar or better accuracy.

1.5.1 Neural Networks and data streams

The first contribution of this dissertation is to study the use of Neural Networks (NN) as an alternative method for processing data streams. How to use them to process data streams is not straightforward due to: 1) the fact that the NN convergence is slow, requiring large amounts of data; and 2) NN are highly sensitive to hyper-parameter configurations such as the depth or the number of neurons, which complicates their deployment in production environments.

In this first contribution, we propose the use of random features in the form of a random projection layer in order to mitigate NN deployment and latency issues. We test the proposal on top of a single layer feed-forward NN, trained with SGD.

We show that NN can achieve good results on data streams classification problems; however they still require some more work to become a feasible method for processing data in real time. We also show that the HT is an easy-to-deploy method, for fast and accurate learning from data streams.

1.5.2 Echo State Hoeffding Tree learning

In this second contribution, we extend two popular incremental tree-based models for data streams, the Hoeffding Tree (HT) and the Fast Incremental Model Trees with Drift Detection (FIMT-DD), in order to capture temporal behaviour. We reuse some of the insights obtained from the first contribution in this dissertation, and propose the use of a reservoir memory that makes use of random features as a recurrent layer that enables capturing temporal patterns. The reservoir's output is then used as the input to a HT or a FIMT-DD. We call this combination Echo State Hoeffding Tree (ESHT).

The ESHT regression capabilities are tested on learning some typical string-based functions with strong temporal dependences. We show how the new architecture is able to incrementally learn these functions in real time with fast adaptation to unknown sequences, and we analyse the influence of the reduced number of hyper-parameters in the behaviour of the proposed solution.

On classification problems, we tested our proposed architecture to learn three well-known data streams datasets. We show that our architecture can in fact improve a single HT. However, our design requires tuning of additional hyper-parameters which makes the proposal not very suitable for production environments, as opposed to other well established methods based on the use of ensembles, such as the Adaptive Random Forest (*ARF*).

1.5.3 Resource-aware Elastic Swap Random Forest

In this third contribution, we present an extension to ARF for processing evolving data streams: Elastic Swap Random Forest (ESRF). ESRF aims at reducing the number of trees required by the state-of-the-art ARF ensemble while providing similar accuracy. ESRF extends ARF with two orthogonal components: 1) a swap component that splits learners into two sets based on their accuracy (only classifiers with the highest accuracy are used to make predictions; the rest are trained as candidates and may be swapped with the others if their accuracy is higher); and 2) an elastic component for dynamically increasing or decreasing the number of classifiers in the ensemble.

The experimental evaluation of ESRF and its comparison with the original ARF shows how these two new components effectively contribute to reduce the number of classifiers up to one third while providing almost the same accuracy. This reduction results in speed-ups, in terms of per-sample execution time, close to 3x. In addition, we perform a sensitivity analysis of the two thresholds determining the elastic nature of the ensemble, establishing a trade–off in terms of resources (memory and computational requirements) and accuracy (which in all cases is comparable to the accuracy achieved by ARF using a fix number of 100 trees).

1.5.4 Ultra-low latency Random Forest

This final contribution presents a high-performance, scalable architecture for designing decision trees and ensemble combinations to tackle today's application domains. The proposed architecture offers ultra-low latency (few microseconds) and good scalability with the number of cores on commodity hardware when compared to other state-of-the-art implementations.

The evaluations show that on an Intel i7-based system, processing a single decision tree is 6x faster than MOA (Java), and 7x faster than StreamDM (C++), two well-known reference implementations. On the same system, the use of six cores (and 12 hardware threads) available allow processing an ensemble of 100 learners 85x faster that MOA while providing the same accuracy.

Furthermore, the proposed implementation is highly scalable: on an Intel Xeon socket with large core counts, the proposed ensemble design achieves up to 16x speed-up when employing 24 cores with respect to a single-threaded execution.

Finally, our design is highly adaptive to different hardware platforms including constrained hardware platforms such as the Raspberry Pi3, where our proposed design achieves similar performance than MOA on an Intel i7-based machine.

1.6 Organization

The rest of this dissertation is organized as follows:

Chapter 2 provides the context of the work presented in this dissertation and the fundamentals for understanding the challenges in learning from data streams. The necessary technical details are also reviewed in this chapter, alongside with a brief overview of other research works related to this dissertation.

Chapter 3 describes the workloads used in the evaluations and provides the information on framework that has been used to evaluate the proposals in this dissertation.

Chapter 4 presents the first contribution of this dissertation, *Neural networks and data streams*. This chapter details the study of neural networks as an alternative method for processing data streams in real time.

Chapter 5 details and evaluates the second contribution of this dissertation, the *Echo State Hoeffding Tree*,s an architecture for real-time classification based on the combination of a Reservoir and a HT decision tree.

Chapter 6 discusses the *Resource-Aware Elastic Swap Random Forest* contribution. We propose and evaluate two methods for reducing the resources needed on ensemble learning: Swap Random Forest (SRF) and Elastic Swap Random Forest (ESRF).

Chapter 7 discusses and evaluates the last contribution of this dissertation: *Ultra-low latency Random Forest*, a high-performance scalable design for decision trees and ensemble combinations that make use of the vector SIMD and multicore capabilities available in modern processors to provide the required throughput and accuracy (in the order of microseconds).

Finally, chapter 8 concludes this dissertation, summarizing its key contributions and results. Also, it presents potential future work and open research lines.

1.7 Publications

The following publications contain the contributions in this dissertation as presented in journals and conferences.

Journal Publications

- **Data Stream Classification Using Random Feature Functions and Novel Method Combinations**
  Journal of Systems and Software, 2016
  Diego Marrón, Jesse Read, Albert Bifet, Nacho Navarro

Conference Publications

- **Echo State Hoeffding Tree Learning**
  The 8th Asian Conference on Machine Learning, 2016, Hamilton, New Zeland
  Diego Marrón, Jesse Read, Albert Bifet, Talel Abdessalem, Eduard Ayguadé, José R. Herrero

- **Low-latency Multi-threaded Ensemble Learning for Dynamic Big Data Streams**
  IEEE International Conference on Big Data 2017, Boston, MA, USA
  Diego Marrón, Eduard Ayguadé, José R. Herrero, Jesse Read, Albert Bifet

- **Elastic Swap Random Forest** (Submitted)
  International Joint Conference on Artificial Intelligence (IJCAI) 2019, China
  Diego Marrón, Eduard Ayguadé, José R. Herrero, Jesse Read, Albert Bifet

Poster Session

- **Random Projection Layer for Neural Networks**
  PhD Poster Session at The Fourteenth International Symposium on Intelligent Data Analysis, 2015
  Diego Marrón, Jesse Read, Albert Bifet, Nacho Navarro

Video Contest

- **Random Projection Layer for Neural Networks**
  The Fourteenth International Symposium on Intelligent Data Analysis, 2015
  https://www.youtube.com/watch?v=ySoaG-rqbr0
  Diego Marrón, Jesse Read, Albert Bifet, Nacho Navarro

2 Preliminaries and Related Work

Learning from data streams can be considered as an infinite dynamic process that encapsulates on a single cycle, the collection of the data, the learning process, and the validation of the learned model. It mainly differs from the traditional supervised machine learning by the fact that instances are not available as a large static dataset. Instead, instances are provided one by one from a continuous data stream that can last potentially forever.

Extracting useful knowledge from a potentially infinite amount of data imposes serious challenges. First, given the vast amount of data arriving, storing it before the processing step is not feasible. Second, when collecting data for long enough periods the relations or patterns learned from data are likely to change as time passes, making old instances to become irrelevant for the current learned model. For example, trends in social networks continuously change due to different reasons. And third, data streams algorithms require sophisticated mechanisms for dealing with noise and contradictory information as they arrive, which adds another dimension to the already complicated task of providing theoretical guarantees on the performance of online learning algorithms.

The stream data mining community has approached the problem from a more practical perspective: algorithms must be designed to satisfy a list of requirements in order to efficiently process data streams. In [58, 14], authors identified the ideal features a data stream algorithm should possess:

1. Should make a single-pass over the data.
2. The model should incrementally incorporate new information without rebuilding the entire model.
3. High speed of convergence.
4. Time for processing each instance is limited.
5. Ideally, CPU and memory consumption must be constant and independent of the number of samples processed.
6. Should react/adapt to changes while they are occurring.

Features 1-5 can in fact be met by variety of learning schemes, including batch learners where batches are constantly gathered over time, and newer models replace older ones as memory fills up [92, 20, 34, 89]. Nevertheless, incremental methods remain strongly preferred in the data streams literature [9, 11, 74] since they usually produce better results [93]. Among them, incremental decision trees are one of the well established methods for processing data streams, being the building block for more powerful methods such as ensemble combinations. Decision trees and their ensemble combinations are the focus of this dissertation. Another popular choice is the k-nearest neighbors (kNN) [100, 5, 73, 65] method, which uses the distance as the metric to find the k-nearest neighbors from the training dataset to a target data point. The original method internally stores instances in order to find the closest neighbors of each input instance which is not feasible in a real time streams setup. Extensions to the original kNN have been proposed for dealing with the inherent performance limitations over a potentially infinite stream [66, 116, 72, 103].

Last feature, feature 6, refers to a change in the function generating the stream. When a persistent change occurs, old instances may become irrelevant or even harmful to the current learned model, degrading its performance or invalidating it. This change is referred to as concept drifting, and data streams which present concept drift are conventionally known as evolving data streams.

This chapter is organized as follows. In Section 2.1, we give the necessary background for understanding concept drifting and the challenges it presents in a data streams setup. Incremental decision and regression trees are introduced in Section 2.2, giving details about the Hoeffding Tree and FIMT-DD, the two methods used in the contributions of this dissertation. Section 2.3 introduces ensemble learning, and give relevant details for the ensemble methods that are also used in this dissertation. Finally, Section 2.4 reviews the current status of Neural Networks applied to data streams, as an alternative methodology to decision trees and ensemble methods.

2.1 Concept Drifting

Learning from data streams is challenging since conventional machine learning methods assume that the data is static (i.e., the data distribution is stationary), while many situations in the real world tend to be dynamic (i.e., non-stationary data distribution). For example, trending topics in social networks or people's opinion about a celebrity/politician change frequently, frauds evolve constantly. These situations imply a change in the target concept (e.g., what was interesting in the past it is not interesting anymore) potentially invalidating old patterns previously learned before the change. This highlights another important aspect when processing data streams: the strong temporal dependence of data, in the sense that only recent data is relevant for the current concept.

A concept drift is a persistent change in the underlying probability distribution generating the data stream, and it should not be confused with outliers (which are transient). Different reasons can cause a change in the concept; however, in practice, the focus of attention is the speed of the change. Depending on that, drifts can be of any of the following types [112]:

- **Abrupt/Sudden**: the concept (data distribution) changes abruptly from one time instance (t) and the next one ($t + 1$). All instances $\in [0, t]$ are generated using concept C_1, and starting from $t + 1$ samples are drawn from concept C_2.

- **Gradual**: the change to the new concept takes a while to complete. While the change is active, the probability of seing samples from C_2 increases while the probability of seing samples from C_1 decreases. This transition is usually monotonic, but not necessarily.

- **Incremental**: change also takes a while to complete, but unlike in gradual drifts, concepts are blended while drift is active. This type of drift is difficult to detect since it can be easily confused with a series of short abrupt drifts. If this drift is active during a short period of time, it is called incremental fast drift; otherwise, it is known as an incremental moderate.

- **Recurrent**: When a previous concept re-appears over time. Note that a previous concept can re-appear suddenly, incrementally or gradually.

The presence of a concept drifting is usually reflected in the model performance: the number of errors gradually or abruptly increase due to the inconsistency between the induced model and newly arriving instances. Therefore, to ensure a proper model adaptation to the current concept, appropriate monitoring of the learning process is necessary.

There are several strategies for dealing with concept drifting [112, 37]. In this dissertation, we use the ADWIN [10] drift detector which provides theoretical guarantees about false positives and negatives.

2.1.1 ADWIN Drift Detector

The Adaptive Windowing (ADWIN) [10] is a concept drift detector that uses an adaptive size sliding window algorithm for detecting changes in data streams. Its main features are: 1) the sliding window is automatically resized depending on the stream characteristic, 2) it provides theoretical guarantees on the ratio of false positives and false negatives when detecting drifts, and 3) it makes no assumptions about the data.

The ADWIN algorithm keeps a sliding window W, with the last n recently received values. Input values can be a binary stream representing errors in classification or a stream of real values representing the loss. The algorithm repeatedly split W into two sub-windows w_0 and w_1 at different points in the window W. For each split point, it computes $\hat{\mu}_{w_0}$ and $\hat{\mu}_{w_1}$ corresponding to the average of the values in sub-windows w_0 and w_1, respectively. If the difference in the averages is above a threshold ϵ_{cut} (i.e., $\mid \hat{\mu}_{w_0}$ - $\hat{\mu}_{w_1} \mid > \epsilon_{cut}$) this indicates a significant drop in performance caused by a change in the error distribution, and a change in the stream is signaled. When a drift is detected, older elements in W are dropped until $\mid \hat{\mu}_{w_0}$ - $\hat{\mu}_{w_1} \mid$ is below the threshold ϵ_{cut}; this causes W to shrink.

The threshold ϵ_{cut} is computed as shown in equation 2.1, where W is the current length of W, and m is the harmonic mean of the two sub-windows averages $\hat{\mu}_{w_0}$ and $\hat{\mu}_{w_1}$ (as shown in equation 2.2). Observe that the only parameter required to configure the ADWIN is $\delta \in (0, 1)$ representing the confidence in applying the cut-off ϵ_{cut}.

$$\epsilon_{cut} = \sqrt{\frac{1}{2m} ln \frac{4W}{\delta}} \tag{2.1}$$

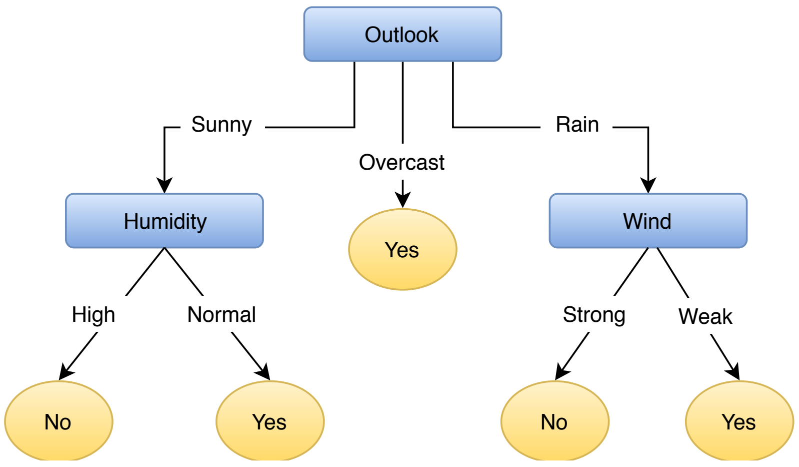
$$m = \frac{1}{\frac{1}{\hat{\mu}_{w_1}} + \frac{1}{\hat{\mu}_{w_1}}} \tag{2.2}$$

ADWIN does not explicitly store all elements in W. Instead, it compresses values using a variation of the exponential histogram [26] requiring only O(logW) memory for storing values, where W is the length of the window. This way, instead of testing W split points for dividing W into w_0 and w_1, ADWIN only needs to test O(logW) split points.

2.2 Incremental Decision and Regression Trees

Decision and regression trees use tree-based data structures for modelling complex decision making. They consist of nodes, branches and leaves as shown in Figure 2.2.1. Each internal node (blue squares) tests on a single attribute (e.g., outlook attribute) in order to chose the appropriate path. Branches are the paths connecting two nodes from two subsequent levels in the tree structure; a node has as many branches as values it can take. Leaf nodes (yellow circles) is where the actual label prediction is done; however, they also contain the necessary information for choosing the next split attribute when necessary. In that case, the leaf node is substituted by an internal node representing the attribute to test on.

Figure 2.2.1: Decition tree example. Square nodes represent internal test/split nodes, circle nodes represent leaf nodes.



Decision trees are built using a divide-and-conquer strategy for partitioning the data into smaller subsets. This partitioning works at attribute (or feature) space, which is recursively partitioned into smaller subsets until a stopping criterion is met. For example, in Figure Figure 2.2.1, the *Humidity* represents the data partition using all instances that has *Outlook=Sunny*, and similarly on the rest of nodes. How an attribute is selected for splitting the attribute space is algorithm-dependent, but in general, it is an expensive operation in term of memory and CPU time.

Incremental tree-based models have been proposed in the literature for classification [108, 109] and for regression [110]. Among incremental models, the Hoeffding Tree (HT) [30] is considered the state-of-the-art single classifier for data streams classification, being widely used in this dissertation. In Chapter 5, we also use the FIMT-DD [62], an incremental regressor tree based on the same principles than the HT. Both are further detailed in the next two subsections.

2.2.1 Hoeffding Tree

The Hoeffding Tree (HT) [30] is an incrementally induced decision tree, and it is considered the state-of-the-art learner for data streams classification. It provides theoretical guarantees that the induced tree is asymptotically close to a tree that would be produced by a batch learner when the number of instances is large enough.

The induction of the HT mainly differs from the induction of batch decision trees in that it processes each instance only once, at the time of arrival (instead of iterating over the entire dataset). Instead of considering the entire training data for choosing the best attribute to split on, the HT states that it may be sufficient only to consider a small subset of the data seen at a leaf node. The Hoeffding Bound (HB) [54] is used to decide the minimum number of instances required with theoretical guarantees. It states that the deviation of an arbitrarily chosen hypotheses with respect the best one is no higher than ϵ with probability $1 - \delta$ (where δ is the confidence level). The very nice property of this test is that it works independently of the data distribution, at expenses of requiring more instances to reach the same level of error than the equivalent distribution-specific tests.

Algorithm 1 shows the HT induction algorithm. The starting point is an HT with a single node (the root). Then, for each arriving instance X the induction algorithm is invoked, which routes through HT the instance X to leaf l (line 1). For each attribute X_i in X with value j and label k, the algorithm updates the statistics in leaf l (line 2) and the number of instances n_l seen at leaf l (line 3).

Splitting a leaf is considered every certain number of instances (*grace* parameter in line 4, since it is unlikely that an split is needed for every new instance) and only if the instances observed at that leaf belong to different labels (line 4). In order to make the decision on which attribute to split, the algorithm evaluates the split criterion function G for each attribute (line 5). Usually this function is based on the computation of the Information Gain, which is defined as:

$$G(X_i) = \sum_{j}^{L} \sum_{k}^{V_i} \frac{a_{ijk}}{T_{ij}} \log(\frac{a_{ijk}}{T_{ij}}) \quad \forall\, i \in N \tag{2.3}$$

being N the number of attributes, L the number of labels and V_i the number of different values that attribute i can take. In this expression T_{ij} is the total number of values observed for attribute i with label j, and a_{ijk} is the number of observed values for which attribute i with label j has value k. The Information Gain is based on the computation of the *entropy* which is the sum of the probabilities of each label times the logarithmic probability of that same label. All the information required to compute the Information Gain is obtained from the counters at the HT leaves.

The algorithm computes G for each attribute X_i in leaf l independently and chooses the two best attributes X_a and X_b (lines 6–7). The Hoeffding Bound is used for testing the hypotheses that $G(X_a)$ and $G(X_b)$ deviate from each other. Therefore, an split on attribute X_a occurs only if X_a and X_b are not equal, and $G(X_a) - G(X_b) > \epsilon$, where ϵ is the Hoeffding bound which is computed (line 8) as:

$$\epsilon = \sqrt{\frac{R^2 ln(\frac{1}{\delta})}{2n_l}} \tag{2.4}$$

being $R = log(L)$ and δ the confidence that X_a is the best attribute to split with probability $1 - \delta$. Since the value of ϵ decays monotonically with the number of instances seen, a leaf node only needs to accumulate examples until ϵ becomes smaller than $G(X_a) - G(X_b)$ for choosing X_a as the splitting attribute. If the two best attributes are very similar (i.e. $X_a - X_b$ tends to 0) then the algorithm uses a tie threshold (τ) to decide splitting (line 9).

Once splitting is decided, the leaf is converted to an internal node testing on X_a and a new leaf is created for each possible value X_a can take; each leaf is initialised using the class distribution observed at attribute X_a counters (lines 11–13).

Leaf Classifiers

Although it is not part of the induction algorithm shown in Algorithm 1, predictions are made at the leaves using leaf classifiers built using the statistics collected in them. Each classifier is trained using only the instances seen at its leaf node, and usually, reuses the same statistics collected for deciding the split attribute.

The Majority Class Classifier (MCC) is the simplest classifier used in the HT, which simply tags the arriving instance with the most frequent label seen in that leaf. It requires no extra statistics other than those already being collected at leaf nodes, and its computational cost is almost negligible.

Another very common classifier is Naive Bayes (NB) [38]. It is a relatively simple method that can reuse the statistics already collected at each leaf node. An advantage of this classifier is that it only requires a small number of training data to estimate the parameters necessary for classification. By making the *naive* assumption that all attributes are independent, the joint probabilities can be expressed as shown in Equation 2.5.

$$p(C_k \mid x) = p(C_k) \prod_{i=1}^{n} p(X_i|C_k) \tag{2.5}$$

Algorithm 1 Hoeffding Tree Induction

```
Require:
    $X$: labeled training instance
    DT: current decision tree
    $G(.)$: splitting criterion function
    $\tau$: tie threshold
    $grace$ : splitting-check frequency (defaults to 200)
    $\epsilon$ : Hoeffding Bound
 1: Sort X to a leaf $l$ using HT
 2: Update attribute counters in $l$ based on X
 3: Update number of instances $n_l$ seen at $l$
 4: if ($n_l$ mod $grace$=0) and (instances seen at $l$ belong to more than 1 different classes) then
 5:     For each attribute $X_i$ in $l$ compute $G(X_i)$
 6:     Let $X_a$ be the attribute with highest $G$ in $l$
 7:     Let $X_b$ be the attribute with second highest $G$ in $l$
 8:     Compute Hoeffding Bound $\epsilon$
 9:     if $X_a \neq X_b$ and $G_l(X_a) - G_l(X_b) > \epsilon$ or $\epsilon < \tau$ then
10:         Replace $l$ with an internal node testing on $X_a$
11:         for each possible value of $X_a$ do
12:             Add new leaf with derived statistics from $X_a$
13:         end for
14:     end if
15: end if
```

It is not not clear which classifier among MCC or NB is the best option. In some situations, NB can initially outperform the MCC, but, as time passes NB can be eventually overtaken by MCC. To address this situation, the Adaptive Naive Bayes HT was proposed [55] which only uses NB classifier when it outperforms MC.

Hoeffding Tree and Concept Drift

Concept drift detection and adaptation requires the use of extra modules (drift detectors), which are not included in the original HT algorithm presented in [30]. HT can adapt to a concept drift by either adapting the tree structure, or by starting a new empty tree.

Adapting the tree structure is done by identifying which sub-tree is affected by the drift. Since a drift in an attribute may invalidate its entire sub-tree, per-attributes drift detectors are used for determining the drifting attribute (the root node of the affected sub-tree). A common approach in data streams is the use of option trees where a new candidate sub-tree is started on the drifting attribute. Both candidate and original sub-trees are trained in parallel until enough evidence is collected for dropping one of then by either confirming the drifting or a false positive. Extensions to the original HT have been proposed using this strategy, such as the Concept-adapting Very Fast Decision Trees CVFDT [59] and the Hoeffding Adaptive Tree (HAT) [11].

Dropping a tree and starting a new empty tree seems to be the simplest strategy: when a drift is detected the entire tree is substituted by a new empty one. This strategy only requires one drift detector monitoring the overall tree performance, which consumes considerably less memory and CPU-time. The drawback is that the there is a lag in time while the tree achieves good performance again. Ensemble combinations used in this dissertation prefer this strategy (as shown in the next section) due to its simplicity and the fact that in ensembles this drawback can be hidden by the rest of the trees.

2.2.2 FIMT-DD

The Fast Incremental Model Trees with Drift Detection (FIMT-DD)[62] is an incrementally induced regression tree which is also based on use of the Hoeffding Bound for growing the tree structure. In the FIMT-DD, the tree structure is a binary tree, i.e. each internal node has two branches.

FIMT-DD uses the same induction strategy as in the HT with proper adaptations for regression, as shown in Algorithm 2. The major difference with respect the HT is that the FIMT-DD incorporates a mechanism for adapting the tree structure in the presence of a concept drift is detected (lines 4–6). The other differences are the necessary changes for enabling the induction process to work in regression problems: 1) how leaf nodes select next attribute to split on (lines 10–14); and 2) predictions are made using a regressor instead of a classifier (not shown in Algorithm 2).

The algorithm computes Standard Deviation Reduction for each attribute in a leaf mode. Let A be the attribute with highest SDR, and B the second one. The ratio r between SDR_A and SDR_B is computed as shown in equation 2.6. The attribute is chosen as for splitting if $r > \epsilon$, where ϵ is the Hoeffding bound already seen in equation 2.4.

$$r = \frac{SDR_B}{SDR_A} \tag{2.6}$$

Prediction at leaf nodes is done using a perceptron with linear output. For each arriving instance an incremental Stochastic Gradient Descent is used for training the perceptron, using all numeric attributes (including those used during the tree traversal).

As mentioned above, in the presence of a concept drift the FIMT-DD starts a new sub-tree for the region where drift is detected (line 5). Both, new and old, sub-trees are grown in parallel until one of them is discarded. The drift detector used in FIMT-DD is the Page-Hinckley [85].

Algorithm 2 FIMT-DD Induction

```
Require:
    $X$: labeled training instance
    DT: current decision tree structure
    $G(.)$: splitting criterion function
    $grace$ : splitting-check frequency (defaults to 200)
    $\epsilon$ : Hoeffding Bound
 1: for each example in the stream do
 2:     Sort X to a leaf $l$ using DT
 3:     Update change detection tests on the path
 4:     if Change is detected then
 5:         Adapt the model tree
 6:     else
 7:         Update statistic in $l$ based on X
 8:         Update number of instances $n_l$ seen at $l$
 9:         if ($n_l$ mod $grace$=0) then
10:             Find two best split attributes ($X_a$ and $X_b$)
11:             if $G_l(X_a) - G_l(X_b) > \epsilon$ then
12:                 Replace $l$ with an internal node testing on $X_a$
13:                 Make two new branches leading to empty leaves
14:             end if
15:         end if
16:     end if
17: end for
```

2.2.3 Performance Extensions

The unfeasibility to store potentially infinite data streams has led to the proposal of classifiers able to adapt to concept drifting only with a single pass through the data. Although the throughput of these proposals is clearly limited by the processing capacity of a single core, little work has been conducted to scale current data streams classification methods.

For example Vertical Hoeffding Trees (VHDT [69]) parallelize the induction of a single HT by partitioning the attributes in the input stream instances over a number of processors, being its scalability limited by the number of attributes. A new algorithm for building decision trees is presented in SPDT [6] based on Parallel binning instead of the Hoeffding Bound used by HTs. [104] propose MC-NN, based on the combination of Micro Clusters (MC) and nearest neighbour (NN), with less scalability than the design proposed in this paper.

Related to the parallelization of ensembles for real-time classification, [79] has been the only work proposing the porting of the entire Random Forest algorithm to the GPU, although limited to binary attributes.

2.3 Ensemble Learning

Instead of focusing on building the most accurate model, ensemble methods [22, 111, 68] combine a set of weak models (base models) for achieving better predictive performance than single models [36, 29]. The main belief in literature is that combining weak models allow wider exploration of different representations or search strategies if there is enough diversity among base learners [88, 94, 83, 84, 49]. Although there is no accepted definition for diversity, the consensus is that ensembles should "spread" the error equally among all learners in the ensemble, so incorrect predictions can be corrected by the majority of the learners in the combination step [70].

Diversity in data streams is usually achieved by introducing a random component during the building of the model, since exhaustive exploration of all possible combinations can quickly become a heavy or an impossible task. Randomization in ensembles can be at different levels:

- *Input*: by creating random subsets from the input data. A well-known algorithm for manipulating the input is the Bagging algorithm [17], which creates a different sub-set for each learner by sampling from the original dataset allowing repetitions

- *Learner*: randomness is introduced in order to improve the exploration of different strategies (sub-sets in this case). For example, Random Forests (RF) [18] use a variation of decision trees that only considers a random subset of the features in the leaf nodes.

The other important ensemble component, as mentioned in [88], is learner's output combination to form the final prediction. A detailed study of different combination methods is presented in [107]. The implicit voting schemes used in the methods we use are the Majority Vote (MV), and the Weighted Majority Vote (WMV) scheme which are very similar: both schemes combine outputs using a fixed simple function (such as aggregation), and then the most voted label is select as the final prediction. The only difference is that MV assume all learners are equally important, while in WMV, a weight decides each learner importance when combining the outputs.

The way predictions can be combined is a direct consequence of how learners interact with each other. According to [43], ensembles can be categorized in the following types depending on their architecture:

- **Flat**: Learners are trained independently on the input data, then their predictions are combined using a simple function (voting scheme). This architecture is the most common architecture used by ensemble methods. Examples of ensembles using this architecture are Online Bagging [86], Leveraging Bagging [13] and Adaptive Random Forest [44].

- **Meta-Learner**: In this category, a meta-dataset substitutes the original input data, and it is used for training meta-learners. The meta-dataset can be created from the output of other learners, such as in Restricted Hoeffding Tree [9], or from data describing the learning problem.
- **Hierarchical**: Members are combined in a tree-like structure, including the cascading or daisy chain. The most notable method for data streams in this category is the HSMiner [16], which breaks the classification problem into tiers in order to prune irrelevant features.
- **Network**: The ensemble is viewed as a graph, which vertices represent learners and the edged represent how learners are connected according to a specific criterion. New connections and vertices can be added at any moment, resembling more to a computer network than to a static graph. For example, SFNC [4] uses a Scale-Free Network model to generate the connections (edges).

The focus of this thesis is in flat ensemble methods, due to their simpler architecture, and the fact that they usually make fewer assumptions about the data distribution. The specific methods used in this dissertation are Leveraging Bagging (LB) and the Adaptive Random Forest (ARF), which are described in the following subsections. Previous to describing LB and ARF, we describe the Online Bagging (OB) method since it is the base for both LB and ARF.

2.3.1 Online Bagging

The Online Bagging (OB) [86] is a flat ensemble method that enforces diversity by introducing randomization at input data, and uses the majority voting scheme for combining learners predictions. OB is the streaming adaptation of the well-known Bagging method introduced in [17].

In the original algorithm, each learner in the ensemble is trained using a subset of the original dataset which is created using the so-called sampling with repetition, i.e., each instance from the original dataset can appear 0, 1, or more in a subset. However, sampling requires storing all instances which is not feasible in the data streams setup.

The OB simulates the sampling with repetition by weighting each input instance instead of storing it. Weights are sampled from a Poisson distribution probability with $\lambda = 1$, and are expected to be 0,1, or > 1 in frequencies 37%, 37%, 26% respectively. Observe that an instance with weight $w = 0$ means the instance is not present in the subset, while $w > 1$ implies that the instance appears more than once.

2.3.2 Leveraging Bagging

Leveraging Bagging (LB) [13] is an extension of the OB ensemble method, that uses a HT as the base learner and includes a drift detection method.

This method leverages the use of the Poisson distribution by using a $\lambda = 6$, altering the output values distribution such as: 0.25% values equal to zero, 45% values lower than six, 16% of values equal to 6, and 39% values greater than 6. By using $\lambda = 6$, the ensemble is using more instances for training, which improves the learning of the ensemble.

LB is an adaptive ensemble method, i.e., it can detect and react to concept drifting. As soon as drifting is detected in any of the *learners*, LB starts a new empty tree in order to replace the one with higher error. LB by design uses the ADWIN drift detector.

2.3.3 Adaptive Random Forest

The Adaptive Random Forest (ARF) [44] was introduced recently as a streaming adaptation of one of the most used machine learning algorithms in the literature, the Random Forest (RF) [18]. ARF has become a popular ensemble method in data streams due to its simplicity in combining leveraging bagging with fast random Hoeffding Trees, a randomized variation of the HT that adds diversity at the learner level.

The Random Hoeffding Tree splits the input data by only considering a small subset of the attributes at each leaf node when growing the tree. When a leaf node is created, $\lfloor \sqrt{N} \rfloor$ attributes are randomly selected from the N original attributes, and used to decide the next attribute to split on.

At input level, ARF enforces diversity by sampling with repetition using a Poisson($\lambda = 6$), as in LB. At the output level, ARF uses the Weighted Majority Voting scheme in which each classifier is weighted using its accuracy.

To cope with evolving data streams, ARF uses a drift detection method based on a two threshold scheme: 1) a permissive threshold triggers a drift warning, which starts a local background learner; and 2) a second threshold confirms the drift, in which case the background learner substitutes the original learner. Although ARF is not tailored to any specific detector, the preferred (default) option is ADWIN.

2.4 Neural Networks for Data Streams

Most Neural Networks (NN) applications assume stationary data distribution where the training is done in a batch setup using a fixed network architecture (number of layers, neurons per layer). The training step is computationally very costly, which depending on the task to learn, it can take days or even weeks to reach the desired error level.

The Back Propagation (BP) [51, 96] algorithm is widely used for training neural networks in conjunction with the Stochastic Gradient Descent (SGD) [23, 47] for minimizing the error at the output layer. Although SGD can be trained incrementally and has strong convergence guarantees, its main issue is that its convergence is slow, which in the data stream setting contradicts feature 4, that time for processing each instance is limited, mentioned at the beginning of this Chapter.

In the Deep Learning literature, extensions for accelerating the SGD convergence have been proposed, such as AdaGrad [32], RMSProp [106] and Adam [67]. AdaGrad uses a per-element (in the gradient vector) learning rate which automatically increases or decreases depending if an element is sparse or dense respectively. The learning rate is adapted based on the sum of squared gradients in each dimension. RMSProp and Adam use a moving average of the squared gradients for updating a per-element learning rate.

The depth of the network (number of layers) also influences the SGD convergence: shallow networks converge faster than deeper ones. In 2018, an Online Deep Learning framework was proposed [97] that dynamically increases/decreases the number of layers using a shallow to depth approach, using the Hedge algorithm [35] in conjunction with BP in the training process.

Recurrent Neural Networks [53, 41, 52, 28] have a natural ability for capturing temporal dependencies, a desirable feature for processing data streams since they typically present strong temporal dependence. At the moment of starting this dissertation the most interesting RNN for data streams was the so-called Reservoir Computing [99, 76, 77], and in particular the Echo State Network (ESN) [64]: a shallow RNN that aims for a simpler and faster training [78, 64]. As detailed in the next subsection (2.4.1), despite ESNs being more straightforward than other RNNs, they are computationally very cheap and very easy to implement being able to model non-linear patterns [87].

With the recent advances in Deep Learning and Neural Networks, some works have been proposed in the context of online learning and data streams, most of them after the work in this dissertation was started. Generative Adversarial Networks (GANs) [45] were introduced in 2014 and recently become very popular due to their ability to learn data representations and generate synthetic samples that are almost indistinguishable from the original data. In 2017 GANs were proposed in the context of online learning [7] where authors use GANs for storing historical training data instead of explicitly store the data, this way only the model is propagated instead of the actual data. Also, their proposed design can adapt to new classes online by resizing the output layer. However, it is not tested with concept drifting nor any typical data streams dataset. Other works include the use of forgetting mechanisms [39], or are based on Randomized Neural Networks [115] such as [90].

2.4.1 Reservoir Computing: The Echo State Network

The Echo State Network (ESN) uses a *resevoir* unit (Echo State Layer in Figure 2.4.1) which is responsible for capturing the temporal dependencies of the input data stream, allowing the ESN to act as dynamic short-term memory. The ESL is connected to a Single Layer Feed-Forward Network (SLFN), that is trained using the standard Stochastic Gradient Descent (SGD) with Back Propagation.

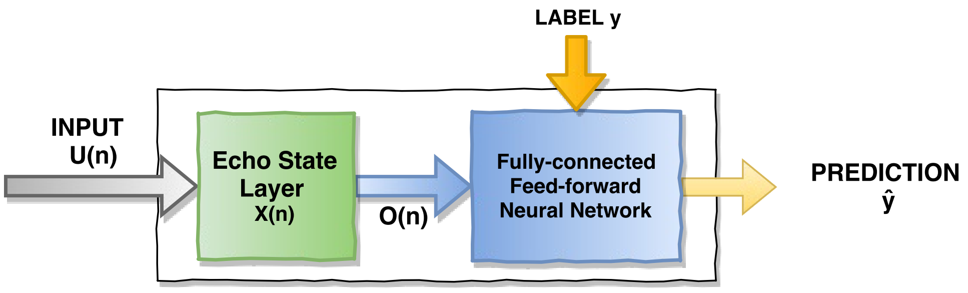


Figure 2.4.1: Echo State Network: Echo State Layer and Single Layer Feed forward Network

Figure 2.4.2 details the ESL. It is a fixed single-layer RNN that transforms time-varying input U(n) to a spatio-temporal pattern of activations on the output O(n). The input $U(n) \in \mathbb{R}^K$ is connected to the echo state X(n) $\in \mathbb{R}^N$ through a weight matrix $W^{in}_{N,K}$. The echo state is connected to itself through a sparse weight matrix $W^{res}_{N,N}$.

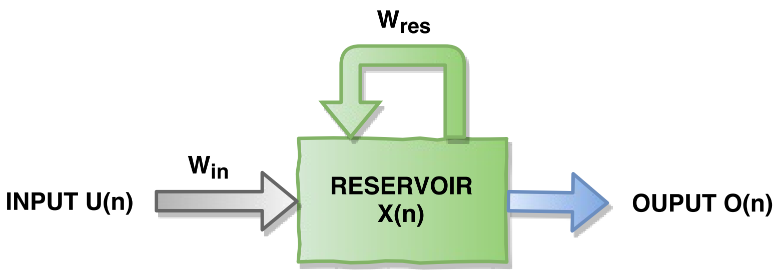


Figure 2.4.2: Echo State Layer

Status update in the ESL involves no derivatives and no error back propagation typically required for training other RNNs. The update is a simple forward step combined with a vector addition, as shown in equations 2.7-2.9. The reduced number of computations needed by the update combined with the ability of modeling temporal dependencies (typically present in data streams), makes the ESL very attractive for real-time analysis.

$$\tilde{w}(n) = \tanh(W^{in}U(n) + W^{res}X(n-1)) \tag{2.7}$$

$$X(n) = (1-\alpha)X(n-1) + \alpha\tilde{w}(n), \tag{2.8}$$

$$O(n) = X(n) \tag{2.9}$$

The hyper-parameter α is used during the echo state $X(n)$ update, and controls how the echo state is biased towards new states or past ones, i.e., controls how sensible the $X(n)$ is to outliers. This resembles the update formula for the momentum explained in Section 4.1.1, where defining $(X) = \nabla C_{t-2}$ as the "new concept to incorporate", and $w_t = X(n)$ as the average, we obtain an almost identical formula; the ESL is also behaving as an exponentially moving average.

Optionally, the input $U(n)$ can be connected to the output $O(n) \in \mathbb{R}^N$ using a weight matrix $W^{out}_{N,K}$. In this work, however, we do not use $W^{out}_{N,K}$ (see Eq. 2.9) since it requires the calculation of correlation matrices or pseudo-inverses which are computationally costly.

The ESL uses random matrices (W^{in} and W^{res}) that, once initialized, are kept fixed (as in the RPL in Chapter 4.1). The echo state $X(n)$ is initialized to zero, and it is the only part that is updated during the execution. Note that the echo state $X(n)$ only depends on the input to change its state. As shown in equation 2.8, calculating X(n) is computationally inexpensive since it only computes a weighted vector addition; the update cost is almost negligible when compared to other RNNs training algorithms.

The ESL needs to satisfy the so-called Echo State Property (ESP): for a long enough input U(n) the echo state X(n) has to wash out any information from the initial conditions asymptotically. The ESP is usually guaranteed for any input if the spectral radius of the ESL Weight Matrix is smaller than unit but is not limited to it: under some conditions the larger the amplitude of the input the further above the unit the spectral radius may be while still obtaining the ESP [114].

Although conceptually simpler than most RNN, computationally cheap, and easier to implement, the ESN still have high sensitivity to hyper-parameter configurations (i.e., small changes to any of them affect the accuracy in a non-predictable way). In order to reduce the number of hyper-parameters, and to accelerate the convergence of the model we propose to replace the SLFN originally included in the ESN with an incremental decision tree, in particular a Hoeffding Tree.

2.5 Taxonomy

Figure 2.5.1 presents a taxonomy for the data streams methods used in this dissertation. Those methods that we specifically extend in our contributions are marked with a yellow box. Regression methods marked with a dashed box are shown for completeness (they are not used in this dissertation). To the best of our knowledge we could not find any *nearest neighbor* proposal

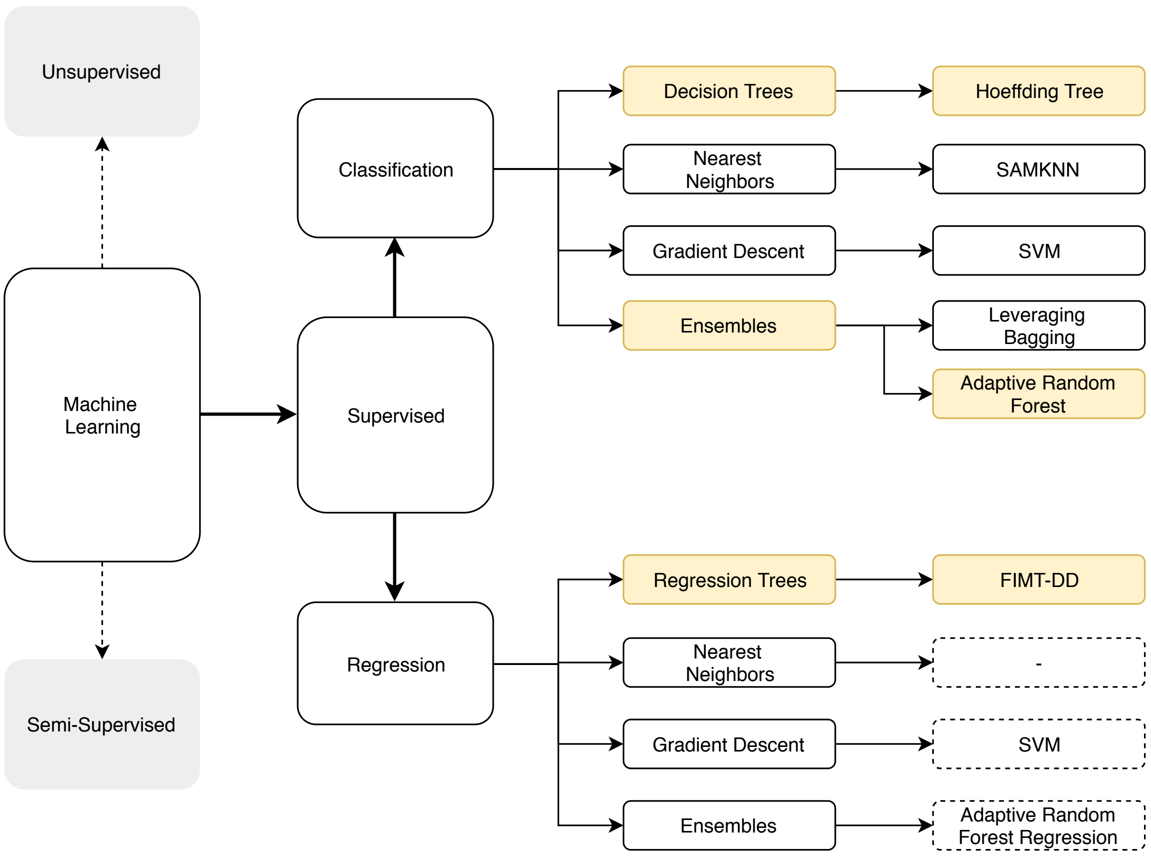


Figure 2.5.1: Taxonomy for data streams methods used in this dissertation. Classification methods: Hoeffding Tree[30], SAMKNN [95], SVM [24], Leveraging Bagging [13] and Adaptive Random Forest [44]. Regression methods: FIMT-DD [62], SVM [46], Adaptive Random Forest regression [42]

for data streams regression. We also show other machine learning areas (grey boxes) for a better perspective of the scope of this dissertation.

3 Methodology

In order to evaluate some of the proposals in this dissertation, we made extensive use of the MOA (Massive Online Analysis) framework [12], a software environment for implementing algorithms and running experiments for online learning from data streams in Java. MOA implements a large number of modern methods for classification and regression tasks, including Hoeffding Tree, K-Nearest Neighbors, Leveraging Bagging and Adaptive Random Forest, among others. Various combinations of these methods were used in the evaluations as a baseline comparison for our proposals.

This chapter also details the datasets used in the evaluations, which comprises real-world and synthetic ones (generated using the synthetic stream generators included in MOA). For each generator, its main characteristics are further detailed in this chapter alongside with the configuration used in MOA for generating each dataset.

This chapter is organized as follows. Section 3.1 describes the MOA framework. Real-world and synthetic datasets used in the evaluations are described in Section 3.2. A table that summarizes all datasets (real-world and synthetic) can be found in Section 3.2.3. Finally, Section 3.3 details the metrics and schemes used to evaluate learning performance.

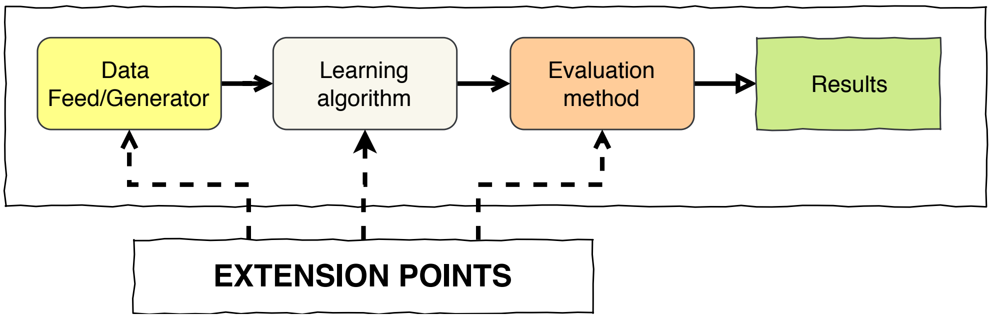


Figure 3.1.1: MOA's workflow

3.1 MOA

Massive Online Analysis (MOA) is the most popular open-source framework for implementing algorithms and running experiments for processing evolving data streams. It is developed at the University of Waikato (New Zealand), and it has a modular architecture that makes it easy to implement new methods and to extend existing ones.

MOA includes an extensive set of algorithms for different types of problems such as classification, regression, clustering or concept drift detection among others. These methods can be directly executed from the Command Line Interface (CLI) or via its Graphical User Interface (GUI). The GUI provides a simple way for configuring each step in the MOA's workflow (detailed in Figure 3.1.1), requiring only to choose the input data, the algorithm, and the evaluation scheme.

An interesting feature in MOA is that it includes several data stream generators that can generate synthetic data on-the-fly. This way, large datasets can be generated with an arbitrary number of instances, features, or labels; the type of concept drifting is also a parameter that can be configured in MOA. The generated data stream can be stored on disk for reuse, or directly used by a learner in which case data is discarded when the execution is completed.

3.2 Datasets

In order to validate the proposed contributions, we make use of different datasets, which were synthetically created or obtained from the real-world. Ten large synthetic datasets have been generated using some of the available synthetic stream generators in MOA; synthetic datasets include abrupt, gradual, incremental drifts and one stationary data stream. Besides, five real-world datasets that have been widely used in several papers on data stream classification are used to conduct the evaluation in this dissertation.

3.2.1 Synthetic datasets

Ten synthetic datasets were generated using known data generators: Agrawal (AGR) [1], LED [19], SEA [102], Radial Basis Function (RBF) [60], Rotating Hyperplane (HYPER) [31] and Random Tree Generator (RTG) [2]. The resulting datasets include different types of concept drift. Gradual and abrupt drift were simulated when using the AGR, LED, and SEA generators. Incremental drift was simulated in the datasets created with the RBF and Hyperplane generators. Finally, when using the RTG generator, no concept drifting was simulated. Next, we detail each generator characteristics, and the CLI configuration we used in MOA to create each synthetic dataset.

Agrawal Synthetic Data Generator

AGR simulates the problem of determining whether a loan should be given to a bank customer or not. Data is generated using one of the ten predefined loan functions, each of which uses a different subset of the features available. Each loan function maps instances to two possible classes.

The data streams generated in this dissertation contain six nominal and three continuous attributes, mapped in two labels, and a 10% of noise (perturbation factor). Concept drift is simulated per each attribute by adding a deviation to its original perturbation factor; deviation is sampled from a random uniform distribution. Two datasets were generated using AGR, each simulating three drifts. This way, ARG_a contains three abrupt drifts, and AGR_g contains three gradual drifts. Configurations used in MOA for generating each dataset are:

ARG_a

```
WriteStreamToARFFFile -f AGR_a.arff -m 1000000 -s
    (ConceptDriftStream -s (generators.AgrawalGenerator -f 1)
    -d (ConceptDriftStream -s (generators.AgrawalGenerator -f
    2) -d (ConceptDriftStream -s (generators.AgrawalGenerator )
      -d (generators.AgrawalGenerator -f 4) -w 50 -p 250000 )
    -w 50 -p 250000 ) -w 50 -p 250000)
```

ARG_g

```
WriteStreamToARFFFile -f AGR_g.arff -m 1000000  -s
    (ConceptDriftStream -s (generators.AgrawalGenerator -f 1)
    -d (ConceptDriftStream -s (generators.AgrawalGenerator -f
    2) -d (ConceptDriftStream -s (generators.AgrawalGenerator )
    -d (generators.AgrawalGenerator -f 4) -w 50000 -p 250000 )
    -w 50000 -p 250000 ) -w 50000 -p 250000)
```

LED Synthetic Data Generator

The LED generator simulates the problem of predicting the digit displayed on a 7-segment LED display. Each feature represents one segment of the display, with a 10% probability of being inverted.

The resulting datasets used in this dissertation contain 24 boolean attributes, where only 7 of them correspond to a segment of the seven-segment LED display (the remaining 17 attributes are irrelevant). Concept drifting is simulated by swapping relevant features with irrelevant ones.

Two datasets were generated using this generator, being the only difference between them in the type of drift simulated. Abrupt concept drift is simulated in LED_a, while LED_g includes gradual concept drifting. Datasets were created using the following MOA configurations:

LED_a

```
WriteStreamToARFFFile -f LED_a.arff -m 1000000 -s
    (ConceptDriftStream -s (generators.LEDGeneratorDrift -d 1)
     -d (ConceptDriftStream -s (generators.LEDGeneratorDrift -d
    3) -d (ConceptDriftStream -s (generators.LEDGeneratorDrift
    -d 5)  -d (generators.LEDGeneratorDrift -d 7) -w 50 -p
    250000 ) -w 50 -p 250000 ) -w 50 -p 250000)
```

LED_g

```
WriteStreamToARFFFile -f LED_g.arff -m 1000000 -s
    (ConceptDriftStream -s (generators.LEDGeneratorDrift -d 1)
    -d (ConceptDriftStream -s (generators.LEDGeneratorDrift -d
    3) -d (ConceptDriftStream -s (generators.LEDGeneratorDrift
    -d 5) -d (generators.LEDGeneratorDrift -d 7) -w 50000 -p
    250000 ) -w 50000 -p 250000 ) -w 50000 -p 250000)
```

SEA Synthetic Data Generator

The default SEA generator uses three features, where one out of the three features is irrelevant. The two relevant features ($f1, f2$) form a 2D space that is divided into four blocks, each representing a concept for simulating concept drifting. New instances are obtained through randomly setting a point in a two-dimensional space, and the label is assigned such that $f1 + f2 \leq \theta$. An instance belongs to class label 1 if the former condition is true, and 0 otherwise. The typical values used for θ are 8 (block 1), 9 (block 2), 7 (block 3) and 9.5 (block 4), and for each block, there is a 10% of class noise.

Two datasets were generated using the SEA generator, one simulating three abrupt drifting (SEA_a), and a second one that simulates three gradual drifting (SEA_g). The following configurations in MOA were used for generating each dataset:

SEA_a

```
WriteStreamToARFFFile -f SEA_a.arff -m 1000000  -s
    (ConceptDriftStream -s (generators.SEAGenerator -f 1) -d
    (ConceptDriftStream -s (generators.SEAGenerator -f 2) -d
    (ConceptDriftStream -s (generators.SEAGenerator )   -d
    (generators.SEAGenerator -f 4) -w 50  -p 250000 ) -w 50  -p
    250000 ) -w 50 -p 250000)
```

SEA_g

```
WriteStreamToARFFFile -f SEA_g.arff -m 1000000 -s
    (ConceptDriftStream -s (generators.SEAGenerator -f 1) -d
    (ConceptDriftStream -s (generators.SEAGenerator -f 2) -d
    (ConceptDriftStream -s (generators.SEAGenerator ) -d
    (generators.SEAGenerator -f 4) -w 50000 -p 250000 ) -w
    50000 -p 250000 ) -w 50000 -p 250000)
```

Radial Basis Function Synthetic Data Generator

The Radial Basis Function (RBF) was devised as an alternative generator for producing complex concept types that are not straightforward to approximate with a decision tree model. It creates a normally distributed hypersphere of examples surrounding random centroid where each centroid has associated a position, a standard deviation, and a class label.

The datasets generated with the RBF simulate incremental concept drifting by moving centroids at a continuous rate, causing instances to potentially belong to a different centroid (thus, different label). RBF_f simulates a fast incremental drift (speed of change set to 0.001), while RBF_m simulates a moderate drift which is ten times slower by using a speed of change of 0.0001. The following configurations in MOA were used to generate each dataset:

RBF_f

```
WriteStreamToARFFFile -f RBF_f.arff -m 1000000 -s
    (generators.RandomRBFGeneratorDrift -c 5 -s .001)
```

RBF_m

```
WriteStreamToARFFFile -f RBF_m.arff -m 1000000 -s
    (generators.RandomRBFGeneratorDrift -c 5 -s .0001)
```

Rotating Hyperplane Synthetic Data Generator

The Rotating Hyperplane generator (HYPER) uses hyperplanes to generate data streams that already contain concept drifting. Incremental concept drifting is simulated by smoothly rotating and moving the hyperplane.

Only one dataset was generated using the HYPER generator. The resulting dataset contains ten attributes and two labels. It contains fast incremental drift, which was achieved by setting the magnitude of change parameter to 0.001. The following configuration in MOA was used for generating this dataset:

HYPER

```
WriteStreamToARFFFile -f HYPER.arff -m 1000000 -s
    (generators.HyperplaneGenerator -k 10 -t .001)
```

Random Tree Synthetic Data Generator

The Random Tree Synthetic Data Generator (RTG) randomly creates an instance which is labeled using a decision tree. The decision tree is completely random: the growing process randomly selects internal nodes and random labels are assigned to leaf nodes.

The dataset generated with this generator contains ten attributes with only two possible labels. Note that the dataset generated with RTG is the only one among the synthetic ones that have no concept drifting. The following MOA configuration was used for generating it:

RTG

```
WriteStreamToARFFFile -f RTG.arff -m 1000000 -s
    (generators.RandomTreeGenerator -o 5 -u 5 -c 2 -d 5 -i 1 -r
    1)
```

3.2.2 Real World Datasets

This subsection describes the five real-world datasets also used in the evaluations: Electricity (ELEC) [48], Forest Covertype (COVT) [15], SUSY [3], Airlines (AIRL) [61], and Give Me Some Credit (GMSC) [25] .

Electricity (ELEC)

This dataset describes electricity demand. It is a dataset for classification which comprises 45,312 instances each of which containing 8 features describing the changes in the price relative to a moving average of the last 24 hours. Each instance is labeled according to the direction of the price change: up or down.

Forest Covertype (COVT)

This dataset describes the forest cover type (the predominant kind of tree cover) for 30x30 meter cells obtained from the US Forest Service of strictly cartographic data. It comprises 581,012 instances, each with 54 attributes. Each of the 7 possible labels corresponds to a different cover type.

SUSY

This dataset has features that are kinematic properties measured by particle detectors in an accelerator. The binary class distinguishes between a signal process, which produces supersymmetric particles, and a background process otherwise. It is one of the largest datasets in the UCI repository that we could find, comprising 5 million instances each with 8 attributes. Instances with two possible labels: 1 for a signal and 0 for a background.

Airlines (AIRL)

This dataset describes the task of predicting whether a given flight will be delayed by looking at the information on the scheduled departure. It is a binary dataset which contains 539,383 instances, each with seven attributes (3 numeric and four nominal).

Give Me Some Credit (GMSC)

GMSC is a credit scoring data set where the objective is to decide whether a loan should be allowed or not, avoiding costly lawsuits. this binary dataset contains 150,000 borrowers, each described by eleven attributes.

3.2.3 Datasets Summary

Table 3.2.1 presents a summary of all datasets used in this dissertation, detailing relevant features of each dataset: the number of samples, attributes per sample, and the number of class labels. For the synthetic datasets, it also details the data generator and the type of drift it contains.

3.3 Evaluation Setup

3.3.1 Metrics

The most common performance metric is the accuracy, which is computed as the ratio of correctly classified instances divided by the total instances seen which also serves for comparing methods since faster learners will obtain a higher ratio. The drawback of this evaluation is that it tends to be pessimistic since early errors of the untrained model count as errors forever. A workaround to this situation is the use of a decaying factor or a sliding window for fading out old predictions.

On classification problems, we use the *accuracy* for comparing the performance between the extended methods against their original version. When

Table 3.2.1: Synthetic (top) and real-world (bottom) datasets used for performance evaluation and comparison.

Dataset	Samples	Attributes	Labels	Generator/DataSet	Drift type
AGR_a	1,000,000	9	2	Agrawal	Abrupt
AGR_g	1,000,000	9	2	Agrawal	Gradual
HYPER	1,000,000	10	2	Hyperplane	Incremental Fast
LED_a	1,000,000	24	10	LED Drift	Abrupt
LED_g	1,000,000	24	10	LED Drift	Gradual
RBF_m	1,000,000	10	5	Radial Basis Function	Incremental Moderate
RBF_f	1,000,000	10	5	Radial Basis Function	Incremental Fast
RTG	1,000,000	10	2	RandomTree	None
SEA_a	1,000,000	3	2	SEA	Abrupt
SEA_g	1,000,000	3	2	SEA	Gradual
AIRL	539,383	7	2	Airlines	-
COVT	581,012	54	7	Forest Covertype	-
ELEC	45,312	8	2	Electricity	-
GMSC	150,000	11	2	Give Me Some Credit	-
SUSY	5,000,000	8	2	SUSY	-

evaluating contributions where we are not modifying the underlying method, accuracy should provide enough evidence that both methods are performing similarly if not identical. Also, faster learners should obtain a higher ratio of correctly classified instances, thus, higher accuracy.

Other common metrics used on binary classification problems are precision, recall or F1 score. Precision metric computes from those instances labelled as *A*, how many were actually label *A*. Recall is the ratio between the number of *A*-labelled instances over the total number of *A*-typed instances in the dataset. F1 score is a combination of precision and recall. Although initially designed for binary classification problems, these metrics can be extended to be used on non-binary data by using the overall combined true/false positive/negatives ratios. We decided to use accuracy for comparing algorithms performance since it's the most used measure in papers on data mining for data streams.

Regression problems often use the *loss* for measuring the error in predictions. The *loss* is computed per instance as the difference between the predicted label ($\hat{y}$) and the correct label (y), i.e., $\hat{y} - y$. For the entire dataset, the *cumulative loss* can be used, which is computed as the summation of all instances loss. In the regression evaluations in Chapter 5, we also use the accuracy, considering an error in classification if the absolute value of the distance is larger than 0.5, i.e., $|\hat{y} - y| > 0.5$. We use a distance of 0.5 since all labels are integer numbers.

Time is an essential metric when evaluating data streams due to response time constrains. We use time as the basic metric in all our experiments for measuring the total CPU time needed to process a dataset. Another time-related measure used in this dissertation is latency, which measures for each arriving instance the time needed to compute its prediction. In Chapter 7, we use the throughput of an algorithm (the inverse of latency) for measuring the number of instances processed per millisecond.

The scalability of our proposed multithreaded ensemble is measured using the relative speed up of increasing the number of threads. This metric is computed as the ratio between the single-threaded execution time divided by the execution time when using more than one threads.

3.3.2 Evaluation Schemes

Evaluation schemes for evolving data streams must consider the strong temporal dependency of data in the sense that only recent data are relevant to the current concept. Consequently, common strategies that split the dataset into test and validation datasets may lose relevant data for the current concept.

The *prequential* [27] or interleaved *test-then-train* evaluation was proposed to fully exploit all data available, using the entire dataset for both testing and training. In order to always test the model on unseen data, each arriving instance is first used for testing (prediction) and then for training. This scheme allows to update the model performance for each arriving instance and obtain the evolution of model performance with time (learning curve).

The *prequential cross-validation* [8] (prequential CV) is an extension to the prequential evaluation which splits test and train data by running multiple experiments; by default, it runs ten experiments in parallel (e.g., ten ensembles in the context of this dissertation). Each arriving instance is used for testing in one run and for training in all the others. This approach maximizes the use of the available data at the cost of redundant work.

In this dissertation, randomization is used at different levels which can influence the final results. In order to minimize the effects of randomization, all experiments are run ten times, and the results are averaged using the arithmetic mean. Observe that the Prequential CV evaluation scheme already includes ten runs.

4 Data Stream Classification using Random Features

In Big Data streams classification, incremental models are strongly preferred since they provide a good balance between accuracy and low response time that is typically required for reacting/adapting to dynamically occurring real-time events. On the other hand, Neural Networks (NNs), are very popular nowadays being widely used in many machine learning areas except for processing data streams, where tree-based models remain the popular choice. In this first contribution, we give a new opportunity to the use of NNs as an alternative method for data stream classification.

NN aim for a better data representation at multiple layers of abstraction, requiring fine tuning for each one of the layers. The typical algorithm used to fine tune the network is the Stochastic Gradient Descent (SGD), which tries to minimize the error at the output layer using an objective function (such as the Mean Squared Error). The error is then back-propagated to previous layers using a gradient vector through the Back Propagation (BP) algorithm. This gradient nature of the BP algorithm makes it suitable to be trained incrementally in batches of size one, similar to the incremental training in online learning.

However, the major impediment for using NN on data streams is related to the fact that they are sensitive to configuration of hyper-parameters such as learning rate (η), momentum (μ), number of neurons per layer, or the number of layers. It is then not straightforward to provide an off-the-shelf NN-based method for data streams.

Furthermore, when collecting real-world data for long enough periods the probability of a change in the underlying concept generating the stream increases with time, degrading the predictive performance of a model; this is known as concept drifting. Typical ways to deal with concept drifting are whether to reset the model or to prune it. Reseting the NN may be worse than ignoring the concept drift due to the slow convergence of SGD-based methods. Alternatively, pruning the network requires to analyse which neurons are redundant or not contributing at all, which is computationally expensive.

This chapter details the first contribution of this dissertation. It highlights the issues of using NN for real-time processing of Big Data streams. We propose the use of random features based on Extreme Learning Machines (ELMs)[56], a recently proposed framework that uses a randomized hidden layer, combined with a fast training algorithm. This way, our design consists on a Single hidden Layer Feedforward Network (SLFN), that uses fixed random weights for connecting the input layer to the hidden layer, while the output layer is the only layer that is trained using the Gradient Descent method. Although the ELM training algorithm is a fast option for training the network it assumes offline training, requiring the computation of the least-squares of a general linear system, which in most cases also implies the computation of the weight matrix pseudo-inverse; it is not clear how to apply the ELM training algorithm for incrementally building the network. Regarding the concept drift, our initial approach is to use a fading factor for previously seen data.

4.1 Random Projection Layer for Data Streams

Inspired in the ELM architecture , this section presents the Random Projection Layer (RPL) for processing data streams (Figure 4.1.1). Our objective was to explore the use of different activation functions, and gradient descent with momentum as an alternative to the originally proposed ELM training algorithm.

The training only updates the weights in the RPL output layer. The input layer uses random weights that, once initialized are never updated, thus avoiding the extra latencies of back-propagating the error across many layers.

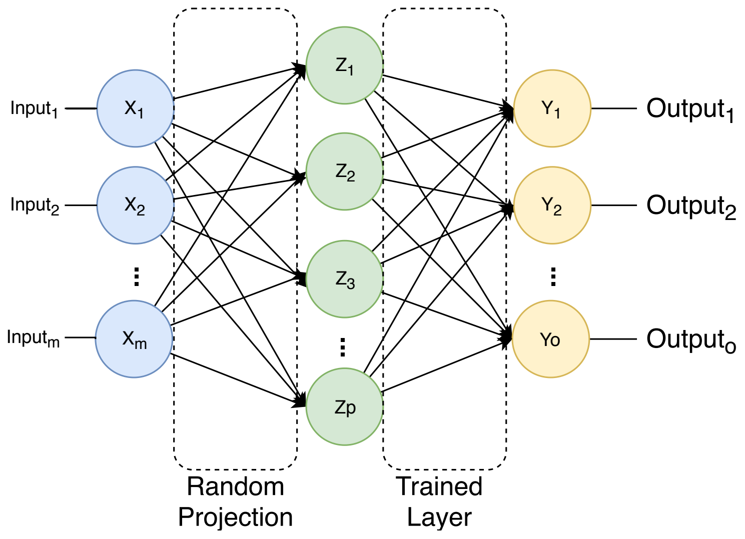
Our design can use different activation functions in the input layer, while in the output layer we only use the sigmoid function. The error at the output layer is measured using the Mean Squared Error (MSE), which in time is used by the SGD for updating the network weights.

4.1.1 Gradient Descent with momentum

In the original Stochastic Gradient Descent (SGD) update formula only the gradient vector ∇C_t is weighted using the learning rate η:

$$w_t = w_{t-1} + \eta \nabla C_t$$

Figure 4.1.1: RPL Architecture. The trained layer uses sigmoid function and MSE as the objective function, while Echo State layer (Random Projection) activation function can vary.



The momentum [91] was proposed for improving the SGD convergence time by adding a weighting factor μ to the standard SGD update formula:

$$w_t = \mu w_{t-1} + (1 - \mu)\nabla C_t \tag{4.1}$$

This simple modification to the SGD accelerates the direction and the speed of the gradient vector ∇C, thus, leading to faster convergence. Compared to the standard SGD update formula, the momentum introduces an opposite relation between past events w_{t-1} and the gradient vector ∇C_t. In other words, defining $\eta = 1 - \mu$ shows the relationship between both terms, therefore, allowing to control the importance of w_{t-1} and ∇C_t; placing more importance in one term reduces the importance of the other one. This is actually behaving like an exponentially moving average, and can be better appreciated expanding the momentum formula through time:

$$\begin{aligned} w_t &= \mu w_{t-1} + (1 - \mu)\nabla C_t \\ w_{t-1} &= \mu w_{t-2} + (1 - \mu)\nabla C_{t-1} \\ w_{t-2} &= \mu w_{t-3} + (1 - \mu)\nabla C_{t-2} \end{aligned}$$

Higher values of μ reduces the importance of ∇C_t since it is equivalent to average over a more significant number of instances. Lower values of μ have the opposite effect, making the sequence closer to ∇C_t, causing fluctuations to the sequence in the presence of noisy derivatives.

Noisy derivatives are the result of rough estimations of the loss function using small batches rather than computing the exact derivative, and sometimes, this estimation makes the gradient vector point in the wrong direction. Thus, momentum (μ) allows controlling the effects of noisy derivatives. When dealing with non-stationary distributions, it may be desirable to control the inertia of past events independently from the gradient vector, thus, using independent μ and η, as denoted in the following expression:

$$w_t = \mu w_{t-1} + \eta \nabla C_t \tag{4.2}$$

Both, μ and η are independent hyper-parameters $\in \mathbb{R}$ in the range $[0, 1]$. Later in this chapter, we provide evidence that this may be a better strategy than using coupled values ($\eta = 1 - \mu$). Using decoupled values allows μ to be used as a simple forgetting/fading factor independently of the learning rate. Note that momentum is no longer acting as an exponentially moving average while still allowing the fading out of old values.

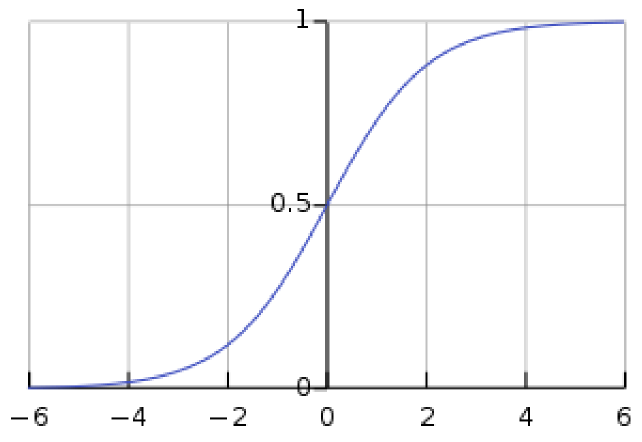
4.1.2 Activation Functions

This subsection details the activation functions used in the evaluations: sigmoid, Radial Basis Function, ReLU, and ReLU-inc. As mentioned above, while the random layer can use different activation functions, the last layer always uses the sigmoid function.

Sigmoid function

The sigmoid activation function is defined in equation 4.3, where $a_k = W_k X$ is the k-th activation function, W_k is the weight $h \times d$ matrix (h output features,

Figure 4.1.2: Sigmoid activation function



d input attributes) including the bias vector, and X is the input to that layer.

$$\sigma(a_k) = \frac{1}{1+e^{-a_k}} \tag{4.3}$$

The sigmoid activation function produces a dense representation since almost all neurons fire in an analog way. This can be costly since all activations will be processed to describe the output.

As can be appreciated in Figure 4.1.2, when $a_k \to \pm\infty$ (x-axis), changes in y-axis ($\sigma(a_k)$) rapidly tend to zero. This means that the gradient vector ∇ tends to zero when $a_k > 2$ or $a_k < -2$; at this point the network stops learning ($\nabla = 0$) or it learns significantly slower ($\nabla \simeq 0$). This is called the gradient vanishing problem.

ReLU and incremental ReLU functions

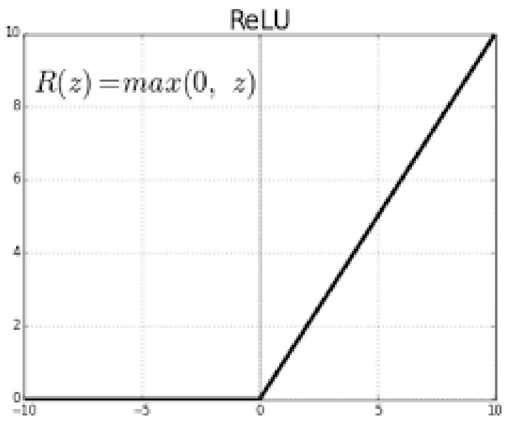
The ReLU activation function is defined in equation 4.4, with a_k defined as above. ReLU allows the network to easily obtain sparse representations; it is expected that 50% of a ReLU output to be zero after a uniform initialization.

$$z_k = f(a_k) = max(0, a_k) \tag{4.4}$$

The implementation of ReLU is very efficient since only a comparison is required (in contrast to the sigmoid function). Figure 4.1.3 shows the ReLU function plot; for negative values of a_k the gradient is zero making neurons stop responding to variations in errors or inputs.

Feature distributions may change with time (due to a concept drifting), and consequently, triggering dense activations. In order to prevent this, ReLU-incremental was introduced [80], a modification to the ReLU that uses the

Figure 4.1.3: ReLU activation function



current mean value of the feature as the threshold in the max function. This mean value is updated for each input instance in order to make it reflect the current feature center, which in time, causes sparse activations. ReLU-incremental is defined as:

$$f(a_k) = max(\bar{a}_k, a_k)$$

Radial Basis Function

The Radial Basis Function (RBF) was proposed in the original ELM paper [57] as a possible activation function for ELM. In this chapter we use an RBF with Gaussian function defined as follows:

$$\phi(x) = e^{-\frac{(x-c_i)^2}{2\sigma^2}} \tag{4.5}$$

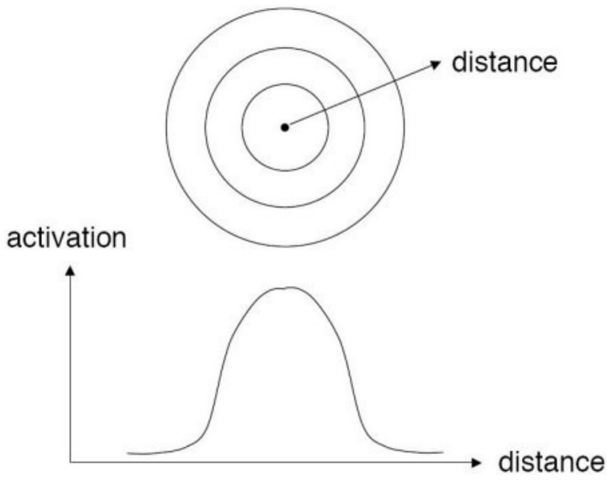
where x is an input instance attribute value, σ^2 is a free parameter, and c_i is a random point. RBF computes the Gaussian distance between the input and the the random point which decays exponentially (Figure 4.1.4).

The notation of equation 4.5 can be simplified by defining a paramater γ as shown in equations 4.6 and 4.7; this notation is the one used later in the evaluation section in this chapter.

$$\gamma = \frac{1}{2\sigma^2} \tag{4.6}$$

$$\phi(x) = e^{-\gamma(x-c_i)^2} \tag{4.7}$$

Figure 4.1.4: RBF activation function using a Gaussian distance.



4.2 Evaluation

This section evaluates the proposed RPL and highlights important issues while using NNs for real-time data stream processing. First, RPL is tested with a variety of activation functions, and different hyper-parameters configurations. Later, we compare RPL with several tree-based and gradient descent based methods. Finally, we compare RPL against a batch version of the SGD method.

The RPL is built processing one instance at a time, using the so-called prequential learning. This is in contrast to typical NNs training, where instances are loaded in batches and the algorithm iterates over them a given number of times until an stop criterion is reached.

Table 4.2.1 summarizes the initialization strategies for the activation functions used in the evaluations (we only show the strategies that achieved best results). Most of the weight matrices are initialized using random numbers with mean μ=0 and standard deviation $\sigma = 1.0$, except for the sigmoid activation function. The bias vector purpose and usage is activation function dependent.

4.2.1 Activation functions

This subsection evaluates the RPL accuracy when using the activation functions presented in Section 4.1.2. In order to test both strategies for momentum (μ) and learning rate (η) mentioned in Section 4.1.1, we tested all possible combinations for both parameters, each in the range $[0.1, 1.0]$ with an increment of 0.1. Regarding the hidden layer, the following sizes were used:

- From 10 to 100 neurons with an increment of 10.
- From 100 to 1,000 neurons with an increment of 100.
- Two additional sizes: 1,500 and 2,000.

Table 4.2.1: Random numbers initialization strategy for the different activation functions

	Weight Matrix		Bias Vector	
Activation	Mean	Std	Mean	Std
Sigmoid	0.0	0.9	0.0	0.2
ReLU	0.0	1.0	0.0	0.1
ReLU-inc	0.0	1.0	0	
RBF	0.0	1.0	γ	

Electricity dataset

Table 4.2.2 shows the best accuracy obtained for the ELEC dataset for each activation function and its hyper-parameters configurations. The sigmoid function obtained the best accuracy (85.34%) using only 100 neurons; ReLU and ReLU-inc difference is marginal, performing not significantly worse than the sigmoid (within 1 percentage point range). The RBF function obtained its best results using the same hyper-parameters configuration, independently of the γ value. When compared to the rest of the activation functions, RBF performed significantly worse.

Figure 4.2.1 shows the accuracy evolution with the RPL size for each activation function. The curve uses the μ and η for which the best accuracy was obtained. The best curve is obtained with the sigmoid activation function. With 50 neurons the accuracy obtained is very close to the best one. Relu and ReLu-inc take longer to achieve an accuracy very close to the best (85.34), while RBF is almost a flat line that slowly benefits from larger RPL sizes.

CoverType dataset

In the COVT dataset evaluation, again, the RBF activation function performed significantly worse than the rest, and it seems that the γ values are not influencing the accuracy. The other three activation functions obtained their best accuracy when using a low learning rate, mid momentum, and ten times more neurons than in the ELEC evaluation. This time the best result was obtained with the ReLU activation function, with marginal difference over the sigmoid (0.06)

Figure 4.2.2 shows the COVT accuracy evolution. The sigmoid achieves 90% of the accuracy with 60 neurons, and very close to the maximum (94.59) with 200. ReLU and ReLU-inc with less than 100 neurons performed worse than the RBF, and required ≥ 800 to achieve an accuracy close to the maxi-

Table 4.2.2: ELEC dataset best results obtained by RPL with different activation functions

Activation	Random Neurons	μ	η	Accuracy(%)
Sigmoid	100	0.3	0.11	**85.33**
ReLU	400	0.3	0.01	84.95
ReLU-inc	200	0.3	0.01	84.97
RBF γ=0.001	2000	0.7	1.01	72.13
RBF γ=0.01	2000	0.7	1.01	72.13
RBF γ=0.1	2000	0.7	1.01	72.13
RBF γ=1.0	2000	0.7	1.01	72.13
RBF γ=10.0	2000	0.7	1.01	72.13

Figure 4.2.1: ELEC Dataset accuracy evolution for the different random layer sizes. This plot used $\mu = 0.3$ and $\eta = 0.11$

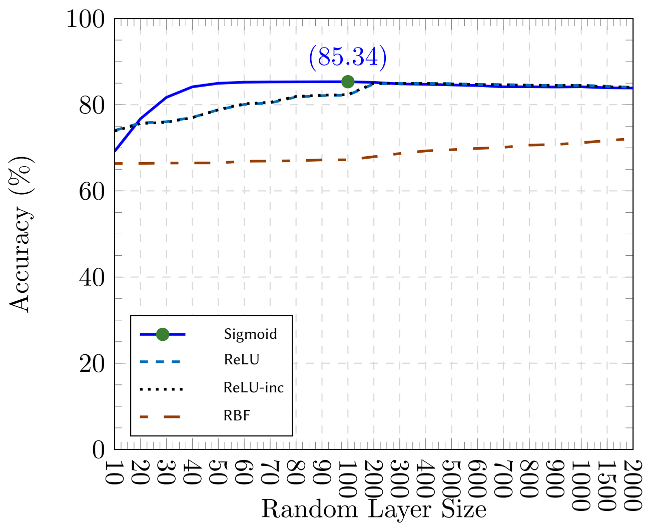


Table 4.2.3: COVT Evaluation

Activation	Random Neurons	μ	η	Accuracy(%)
Sigmoid	1000	0.4	0.11	94.45
ReLU	2000	0.4	0.01	**94.59**
ReLU-inc	2000	0.4	0.01	94.58
RBF γ=0.001	90	0.9	1.01	73.18
RBF γ=0.01	90	0.9	1.01	73.18
RBF γ=0.1	90	0.5	1.01	73.18
RBF γ=1.0	90	0.8	1.01	73.18
RBF γ=10.0	90	1.0	1.01	73.18

mum. The RBF function seems to perform equally independently of the random layer size.

SUSY dataset

The last dataset evaluated is SUSY, which exhibits opposite trends with respect the two previous datasets as shown in Table 4.2.4. RBF obtained the best accuracy, while the sigmoid function performed significantly worse than the sigmoid and ReLU ($\simeq$ 10 points). Again, the γ value on the RBF is not influencing the accuracy.

As shown in Figure 4.2.3, the sigmoid curve is always worse than the RBF, achieving its maximum peak with 20 neurons. The sigmoid with ≥ 40

Figure 4.2.2: COVT Normalized Dataset

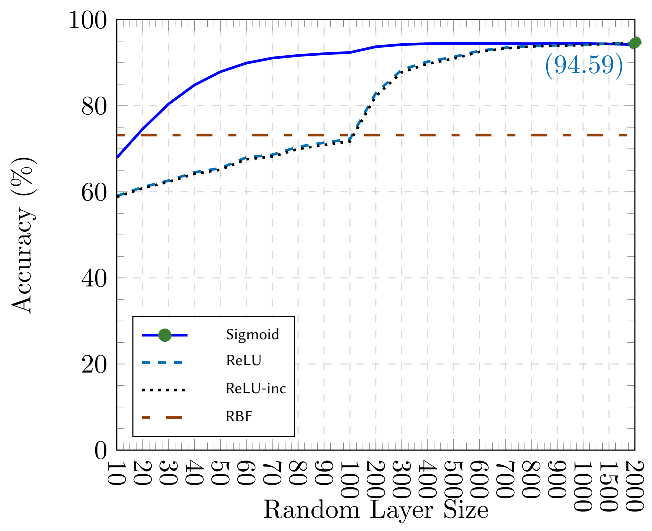


Table 4.2.4: SUSY Evaluation

Activation	Random Neurons	μ	η	Accuracy(%)
Sigmoid	20	1	0.61	67.28
ReLU	20	1	0.61	74.84
ReLU-inc	20	1	0.91	74.80
RBF γ=0.001	600	1	0.71	**77.63**
RBF γ=0.01	600	1	0.71	**77.63**
RBF γ=0.1	600	1	0.71	**77.63**
RBF γ=1.0	600	1	0.71	**77.63**
RBF γ=10.0	600	1	0.71	**77.63**

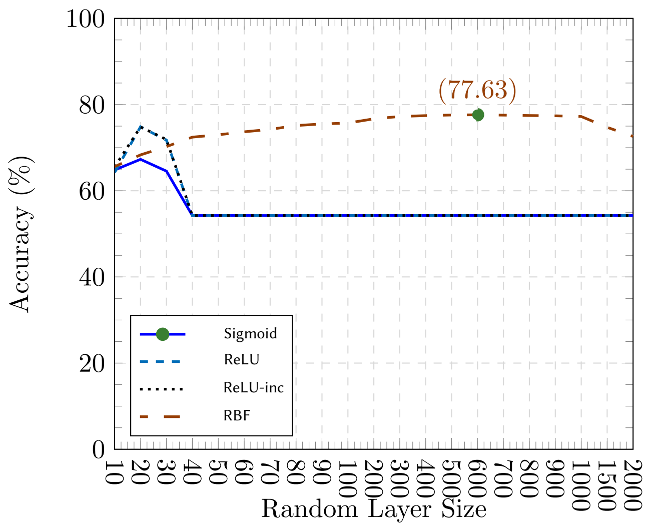
neurons accuracy drops significantly staying constant as the size grown. Both ReLU and ReLU-inc present a similar behaviour.

Discussion

In general, the RBF activation function seems not very sensitive to different values of γ parameters. Figures 4.2.1, 4.2.2 and 4.2.3 show that RBF performance is very stable in all datasets even if the results are not very good in some cases (ELEC and COVT).

On the other hand, there is no clear correlation on how hyper-parameters are affecting learning. The empirical tests suggest that using independent hyper-parameters for momentum (μ) and learning rate (η) is a good strategy, since none of the best results presented the relation $\mu = 1 - \eta$ shown in section

Figure 4.2.3: SUSY Dataset



4.1.1. Using fixed values for (μ) and (η) averages across the entire dataset, and in the presence of a concept drift for example, it may be preferable to control dynamically (μ) and (η) in order to properly adapt to the new underlying data distribution. Dynamically controlling (μ) and (η) is out of the scope of this work.

Testing many configurations in order to obtain the best one is not a feasible strategy in online learning for two reasons: there is no clue when the underlying data distribution is going to change; and the model has a finite time to learn from the data.

4.2.2 RPL comparison with other data streams methods

This subsection compares, in terms of accuracy, the best RPL results obtained in the previous section with other well-known streaming methods available in the MOA framework. Reference methods used are: the Hoeffding Tree (HT), Leveraging Bagging (LB), K-nearest neighbours (kNN) and Stochastic Gradient Descent (SGD) without hidden layer. All methods were tested using the default values in MOA, except for kNN that uses a buffer size of 5,000, and LB that uses 10 learners.

Table 4.2.5 details the accuracy obtained for each method and dataset. The best performing method on average is LB, while our proposed RPL achieved the second best average rank.

In the COVT evaluation, RPL obtained the best accuracy outperforming by more than two points the kNN algorithm. Despite RPL obtaining the second best accuracy in the ELEC dataset, the difference with the best (LB) is more

Table 4.2.5: RPL accuracy (%) comparison against other popular data streams methods.

	HT		SGD		kNN		LB		RPL	
Dataset	Acc(%)	rank	Acc (%)	rank	Acc (%)	rank	Acc (%)	rank	Acc (%)	rank
ELEC	79.2	3	57.6	5	78.4	4	**89.8**	1	85.3	2
COVT	80.3	4	60.7	5	92.2	2	91.7	3	**94.59**	1
SUSY	78.2	2	76.5	4	67.5	5	**78.7**	1	77.63	3
Average	79.23	3.00	64.93	4.67	79.37	3.67	86.73	**1.67**	85.84	2.00

Table 4.2.6: SGD Batch vs Incremental

	Batch				Incremental					
	SGD				SGD		RPL		LB	
Dataset	iterations	neurons	Acc(%)	Rank	Acc(%)	Rank	Acc(%)	Rank	Acc(%)	Rank
ELEC	100	100	85.149	3	57.6	4	85.33	2	**89.8**	1
COVT	10000	2000	88.654	3	60.7	4	**94.59**	1	91.7	2
SUSY	1000	100	76.675	3	76.5	4	77.63	2	**78.7**	1
Average	-	-	83.49	3	64.93	4	85.85	1.67	86.73	1.33

than 4 points. In the last dataset (SUSY), RPL performed relatively poorly being outperformed by a single HT and LB, with a difference of 1.07 points in the worst case (LB).

The SGD method performed poorly in the ELEC and COVT compared to a single HT. The main reason is that tree-based methods require less instances than SGD based methods to reach competitive accuracy. On datasets with low number of instances such as ELEC (45k instances), a single pass on the data is not enough for the SGD method to achieve a good accuracy (only 57.6), specially if compared with HT (79.2). In the next section we show that iterating over the dataset reduces this gap. On datasets with large number of samples, such as SUSY (5M instances), the gap narrows significantly even when only a single pass on thef data is done.

RPL significantly improves SGD in all datasets. Note that RPL is the result of adding a single random layer to the SGD method.

4.2.3 Batch vs Incremental

This last section compares the incremental and batch vesions of the SGD-based methods. The batch version iterates up to 20K times over the dataset, while the incremental version makes a single pass over the data. The same values for μ, η and hidden layer size as in Section 4.2.1 were tested on the batch SGD.

The best results obtained by the SGD-batch and its configurations are summarized in Table 4.2.6. Note that since SGD has no hidden layer, all SGD results in this section were achieved using the sigmoid activation function.

Table 4.2.6 compares in terms of accuracy the Incremental and batch version of the SGD along with RPL results. As stated in section 4.2.2, a single pass over the data achieves good accuracy only when the number of instances is large (the SUSY dataset). Results in Table 4.2.6 show that iterating over a dataset improves the SGD accuracy, specially on those datasets with fewer instances such as ELEC.

Adding the RPL to the SGD improves the network accuracy, making it performing similarly to the batch version. The advantage is that RPL makes a single pass over the data leading to a faster learning, removing the need for iterating over the data. The drawback is that RPL still requires tuning other hyper-parameters such mas momentum (μ), hidden layer size or learning rate (η).

4.3 Summary

The random layer can turn a simple gradient descent learner into a competitive method for continual learning of data streams. Results shown in this chapter suggest that having independent values for momentum (μ) and learning rate (η) is a better strategy for processing data streams compared to using the usual coupled $\mu = 1 - \eta$. The momentum also acts as a forgetting mechanism.

RPL still requires hyper-parameters to be tuned, and it adds one extra hyper-parameter when using independent values for μ and η. RPL, and NNs in general, have proven to be very sensitive to hyper-parameters configurations, requiring many combinations to be tested in order to achieve competitive accuracy. Even when choosing the best configuration for a dataset, RPL could only outperform one of the streaming methods evaluated. And it has been observed that in one dataset the difference can be larger than 4 points.

This chapter highlights important issues of using NNs for data streams classification. How to overcome these issues is out of the scope of this dissertation. Given the results obtained in the evaluations, we decided to go in the direction of using a hybrid alternative combining neural network and tree-based methods contributing with the proposal that is presented in the next chapter.

5 Echo State Hoeffding Tree Learning

This chapter details the second contribution of this dissertation. We focus on improving the Hoeffding Tree (HT) by proposing an extension that is able to capture the strong temporal dependencies that are typically present in data streams: the Echo State Hoeffding Tree (ESHT). We propose the use of a hybrid approach that combines a recurrent neural network (the Echo State Network) with an incremental decision tree or regressor tree depending on the learning task. This combination takes advantage of the good temporal properties of recurrent neural networks for improving single-tree predicting performance.

5.1 The Echo State Hoeffding Tree

In this section we present the Echo State Hoeffding Tree (ESHT), our new approach to learn from data streams with strong temporal dependencies. We propose a hybrid approach that combines the ESL ability to encode time dependencies in the ESN, with a very efficient incremental decision tree. On classification problems the incremental decision tree used is the Hoeffding

Tree. On regression problems we use an HT regressor, the FIMT-DD [62]. The proposed architecture is shown in Figure 5.1.1.

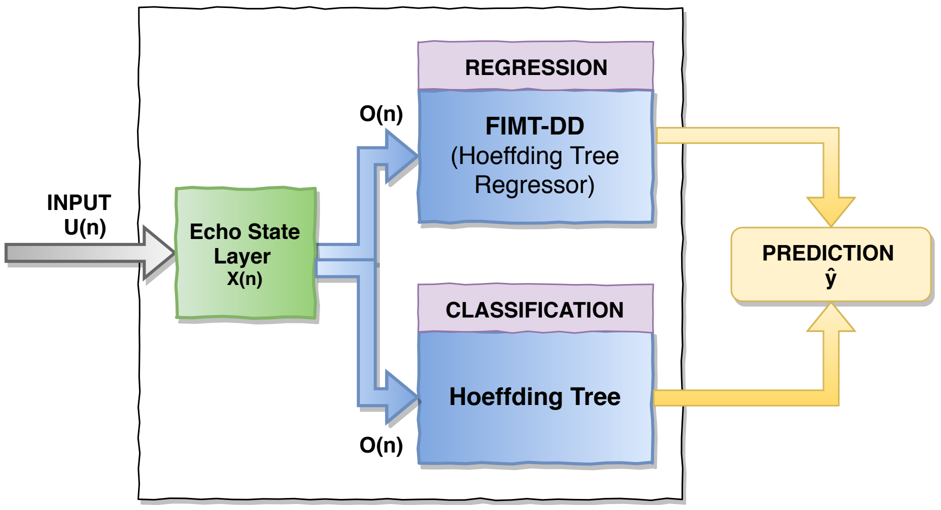


Figure 5.1.1: Echo State Hoeffding Tree design for regression (top blue box) and classification (bottom blue box)

The ESHT uses the same ESL as decribed in Section 2.4.1. The hyper-parameters required to configure the ESL are: α in Eq. 2.8, number of neurons, and density of the sparse matrix $W^{res}_{N,N}$ in Eq. 2.7 (in this work, density $\in (0, 1.0])$. Regarding the number of neurons in the echo state $X(n)$, [75] states as a generic rule to set it up proportional to the number of time steps an input should be remembered.

As can be observed in the architecture in Figure 5.1.1, our proposal replaces the single layer feed-forward NN (SLFN) in the original ESN proposal with a Hoeffding Tree (for classification) or a FIMT-DD (for regression).

The main advantage of using either a HT or a FIMT-DD is that they require less samples than a NN to achieve good accuracy. In addition, they are easier to deploy since they do not require the configuration of hyper-parameters, reducing the deployment complexity of the proposed architecture.

5.2 Evaluation

The capabilities of the proposed architecture are tested on both regression and classification problems. The regression version of ESHT is evaluated with typical string-based functions with strong temporal dependences. The new architecture is able to incrementally learn these functions in real time with fast adaptation to unknown sequences and analyses the influence of the reduced number of hyper-parameters in the behaviour of the proposed solution. On classification problems, we show that the temporal capabilities of ESHT improve the accuracy of a single HT on the same datasets that we also used in Chapter 4.

5.2.1 Regression evaluation methodology: learning functions

As a proof of concept, we propose the ESHT to learn functions typically implemented by programmers. In the evaluations we use what we call a *module* (Figure 5.2.1) that is composed of a label generator and an ESHT. A module learns one function by only looking at its inputs and output (no other information is provided to the ESHT) in real time.

The label generator uses the function we want to learn to label the input. The input to the ESHT can be randomly generated or read from a file, and can be a single integer or a vector. Both input and label are forwarded to the ESHT.

In this work we use only one module in the evaluations, but modules could be combined to solve complex tasks the same way programmers combine functions when writing programs. A potential application of this methodology is to treat programming as a black box: we could write the tests for a function and use the ESHT to learn the function instead of implementing it. This way, scaling computations is a matter of scaling a single model.

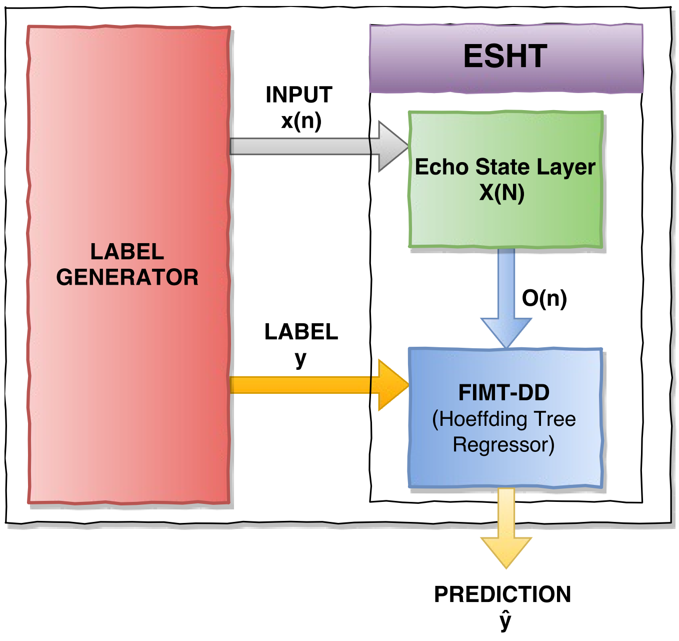


Figure 5.2.1: Module internal design: label generator and ESHT

5.2.2 Regression evaluation

This section evaluates the behaviour of the proposed ESHT architecture for learning three character-stream functions: *Counter*, *lastIndexOf* and *emailFilter*. Function *Counter* counts the number of elements that appear in the input between two consecutive zeros. Function *lastIndexOf* outputs the number of elements in the input since we last observed the current symbol. In other words, it counts the number of elements between two equal symbols in the stream (i.e. use one *Counter* for each symbol). Finally, *emailFilter* is a function that detects valid email addresses in a character stream.

In order to understand the behaviour of ESHT we study the effect of its two hyper-parameters: α in Eq. 2.8 and the *density* of the sparse matrix $W^{res}_{N,N}$ in Eq. 2.7. We use *Counter* and *lastIndexOf* functions for this purpose. In both evaluations we fix the number of neurons to 1,000. In the *Counter* evaluation we test combinations of α and density in the range $[0.1, 1.0]$ in steps of 0.1. In the *lastIndexOf* evaluation we use the outcomes to test only some combinations of α in the same range.

We use *emailFilter* function to compare the behaviour of the proposed ESHT architecture with a FIMT-DD regressor tree, a standard fully-connected feed-forward NN and the ESN.

Two metrics are used for the purposes of evaluating the behaviour of ESHT, both derived from the *errors* detected in the output: *cumulative loss* and *accuracy*. An error occurs when the output is incorrectly classified, i.e. when $|y - \hat{y}| >= 0.5$, being $\hat{y}$ the predicted label and y the actual label; we use a distance of 0.5 since all labels are integer numbers. The *cumulative loss* shows the accumulated $|y - \hat{y}|$ for all the incorrectly classified inputs. And *accuracy* shows the proportion of correctly predicted labels with respect to the number of inputs.

Since ESHT is proposed for real-time analysis, it is also important to analyze the number of iterations that are needed in order to correctly output the correct label for a previously seen sequence. For example, for the *Counter* function evaluation, when we say that ESHT needs to observe a sequence two times, it means that at the third time the ESHT observes that sequence it outputs its correct length.

Counter

Two variations of the *Counter* function have been implemented (shown in Figure 5.2.2). In *Opt1* the label for each symbol in the input stream is the number of symbols since the last zero appeared (and 0 when the zero symbol appears); in *Opt2* the label is different than 0 only when zero appears in the input stream, returning in this case the number of symbols since the previous zero appeared.

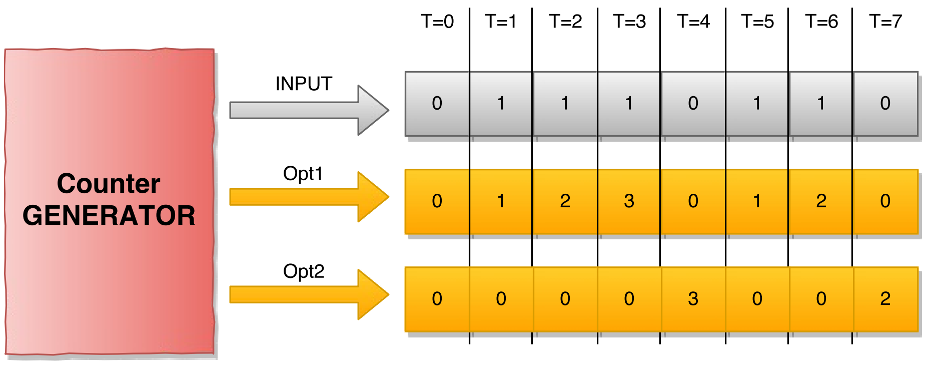


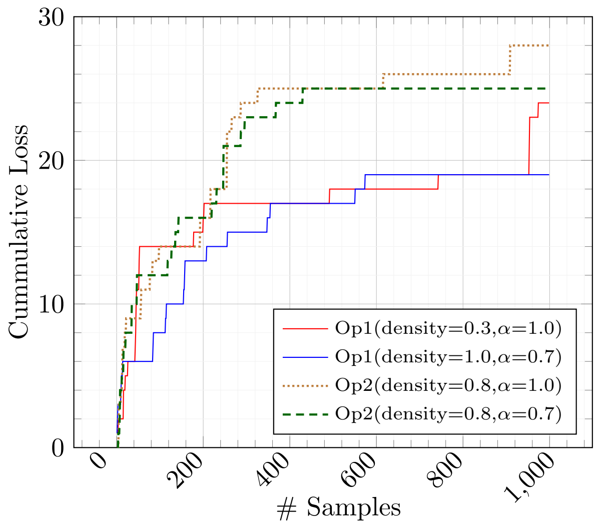
Figure 5.2.2: *Counter* generator functions

The input stream is a random sequence of $0/1$ symbols generated following a normal distribution. We use it to analyze the influence of parameters α and *density* on the *loss* and *accuracy* mentioned in the previous section for an input of 1,000 samples.

Figure 5.2.3 shows the evolution along time (number of input symbols) for the cumulative *loss* (top) and *accuracy* (bottom), for each *Counter* option and for two different combinations of parameters α and *density*. From a visual inspection of the output generated, the first conclusion that we obtain is that ESTH is able to learn possible sequences of the input symbols after seeing them two or three times.

The first observation from the cumulative *loss* plot on the top is that the *errors* and *loss* rapidly decrease with time. This is the expected behaviour for a randomly generated sequence that follows a normal distribution: the more samples we generate the less chance of an unseen sequence (giving ESHT ability to learn already seen sequences). From the *accuracy* plot on the bottom of Figure 5.2.3 we conclude that an accuracy of 0.9 is achieved with only a few hundred samples (200 in this specific configuration of the α and *density* parameters); in other words, almost all loss is incurred on the first few hundreds of samples, and after this, the loss stabilizes being *opt1* the one where loss stabilizes faster. Some of the results have a transient accuracy of 1 for the first item; this is due to ESHT always outputs a zero for the first element (which is the label of the first item in the input stream, as shown in Figure 5.2.2).

Figure 5.2.4 shows the influence of the α (top) and *density* (bottom) parameters in the *accuracy* of the counter *opt1*. In both plots, the horizontal axis shows the variation of one of the parameters while the box plot shows the variation of the other parameter (with values inside the box that have an accuracy with the standard deviation). The plot at the top shows that there is a monotonic growth of the *accuracy* with parameter α; lower values for α place relatively more importance on older reservoir states (see Eq.2.8), which



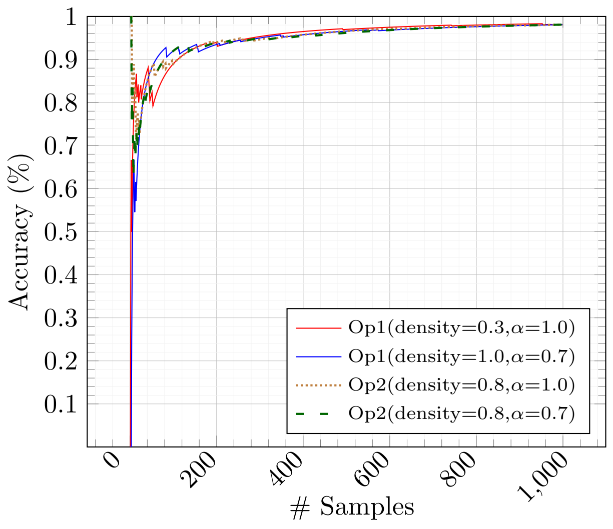
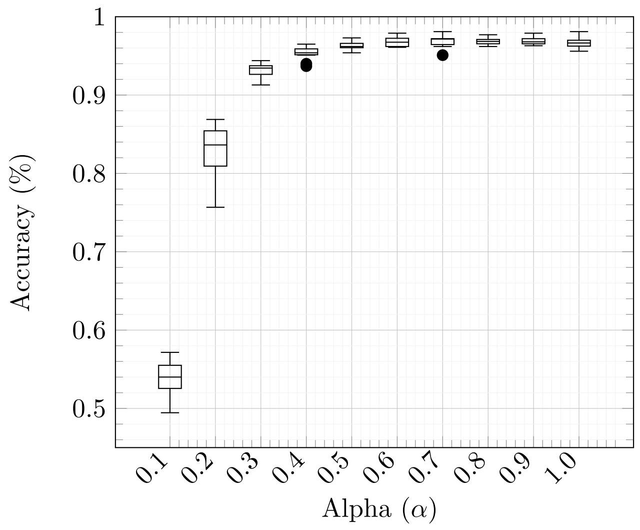


Figure 5.2.3: Cumulative *loss* (up) and *accuracy* (bottom) on the Counter stream.

has the effect that it takes longer for the model to learn new sequences. In this same plot, the influence of *density* seems to have a less relevant influence. In fact the plot at the bottom shows that there is no clear correlation between *density* and *accuracy*. The outliers in that plot correspond to the low values of α that were already commented in the previous plot.

lastIndexOf

Figure 5.2.5 shows the output of the *lastIndexOf* function (which is how this function is known to Java programmers). Given a sequence of input symbols,



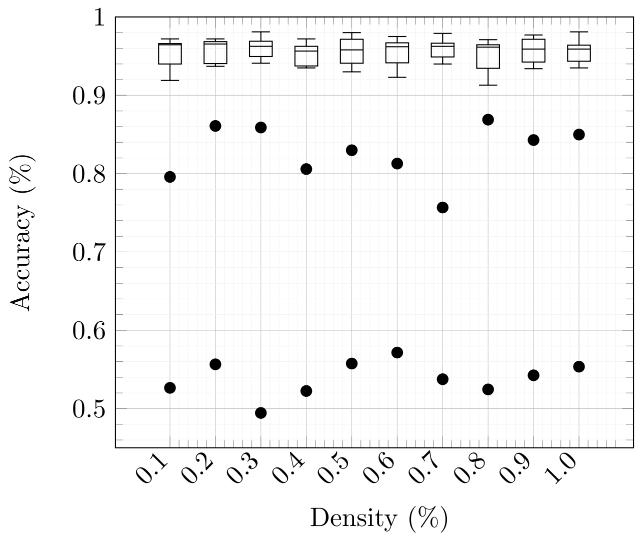


Figure 5.2.4: Influence of parameters α and *density* on the *Counter* stream. In each figure, the box plot shows the influence of the other parameter.

the function returns for each symbol the relative position of the last occurrence of the same symbol (i.e. how many time steps ago the symbol was last observed). Note that for each time-step all symbols but the current one are one step further, thus, generating a highly dynamic output.

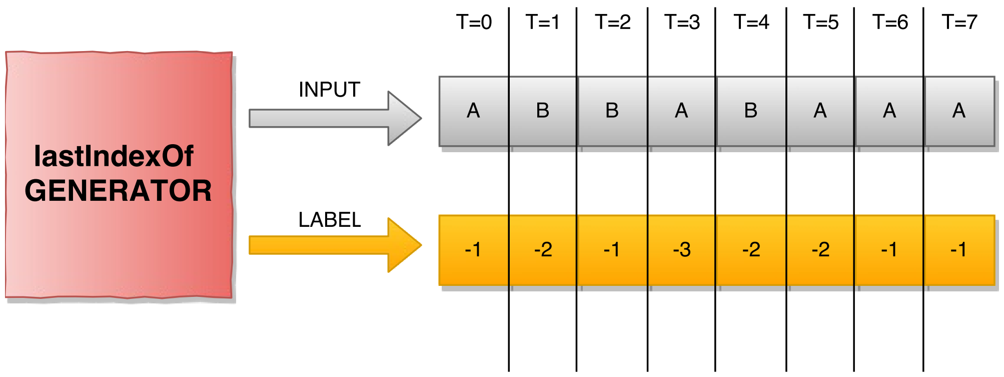


Figure 5.2.5: *lastIndexOf* generation function

The input stream is a sequence of symbols of an alphabet randomly generated following a normal distribution. Sequences of up to 10,000 samples and alphabets of 2, 3 and 4 symbols have been used to perform the evaluation of ESHT in terms of *accuracy*.

As the number of symbols in the alphabet grows more samples are needed to learn a pattern. Consequently, the number of combinations grows exponentially with the number of symbols. Figure 5.2.6 shows this trend for two different pairs of values α and *density*. Alphabets with 2 and 3 symbols are relatively simple to be learnt (the ESHT achieved 80+% accuracy with only 1,000 samples) while with a 4-symbols alphabet the ESHT needed 10,000 samples to achieve 75% accuracy.

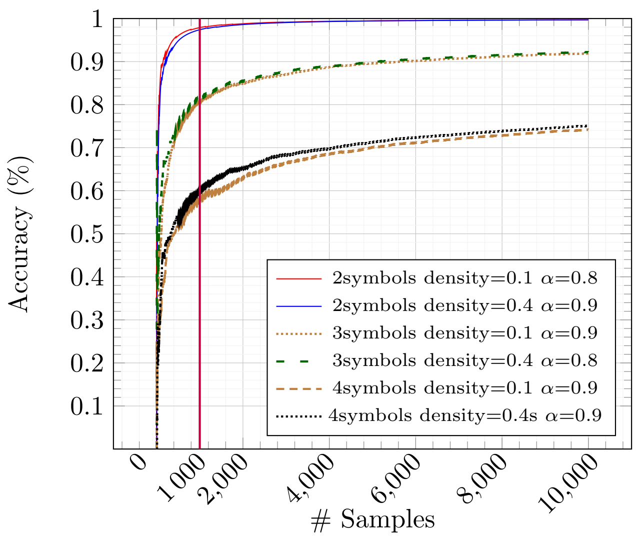


Figure 5.2.6: Accuracy for the *lastIndexOf* function for alphabets with 2, 3 and 4 symbols.

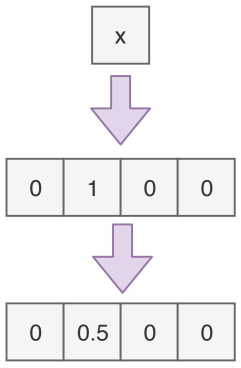


Figure 5.2.7: Encoding *x* symbol as a vector

From a visual inspection on the ESL output we observed that a relatively small number of samples were needed to saturate the output signal. To delay this saturation we decided to use a vector of features (one element for each symbol) as the input instead of a scalar value. The position corresponding to the current symbol index is set equal to *0.5* and the rest equal to zero. Figure 5.2.7 shows how the x symbol is encoded as input vector. This way, the input signal to the FIMT-DD has different levels (in contrast to the saturated signal observed when using a scalar input). Figure 5.2.8 shows the improvement achieved by using a vector instead of scalar input for different values of the α hyper-parameter. For the rest of the evaluations in this subsection we will use the vector input.

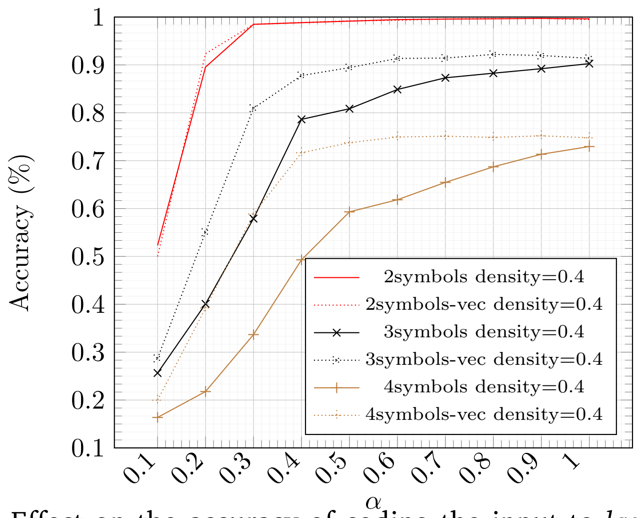


Figure 5.2.8: Effect on the accuracy of coding the input to *lastIndexOf* as a scalar or a vector of features (density=0.4)

Figure 5.2.9 shows the influence of the α and *density* hyper-parameters on the *accuracy*. In both plots, the horizontal axis shows the variation of one of the parameters and different lines are used to show the variation of the other parameter. From the plot at the top it is clear the monotonic growth of the *accuracy* with parameter α. It can be seen in the bottom plot that the influence of *density* seems to have a less relevant influence if the value of $alpha$ is correctly set. In fact the plot at the bottom shows that there is no clear correlation between *density* and *accuracy*. Similarly, one could predict a similar conclusion when changing the number of neurons in the ESL.

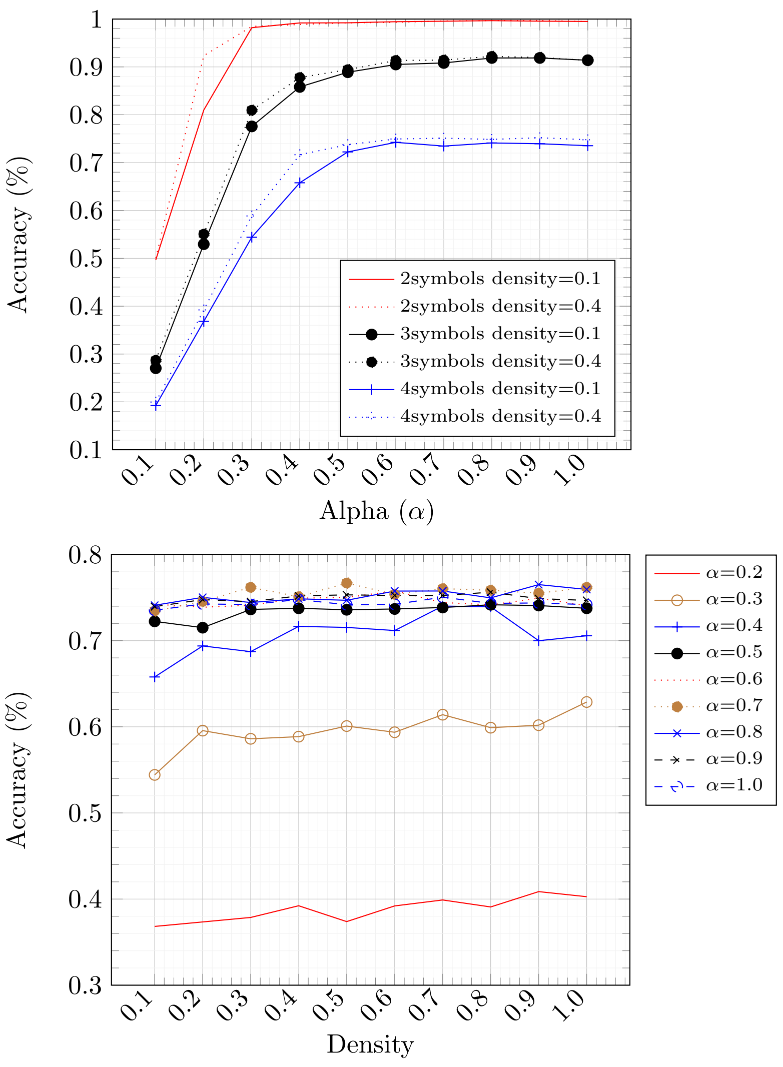


Figure 5.2.9: Effect of $alpha$ and *density* on the accuracy for *lastIndexOf*

emailFilter

The *emailFilter* function labels the input stream with the length of the email address detected or *0* otherwise (including wrong formatted email address).

For the evaluation of the ESHT and comparison with previous proposals we use a synthetic dataset based on the *20 newsgroups* dataset, that comprises around 18,000 newsgroups posts on 20 topics [71]. We extracted a total of 590 characters and repeated them eight times. Each repetition, or block, contains 11 email addresses and random text (including wrong formatted email addresses) at the same proportion. The resulting dataset has a label balance of 97.8% zeros.

Based on the conclusion from *lastIndexOf* evaluation, we decided to represent the input as a vector of features, one for each symbol in the alphabet. However, this would require a vector for an ASCII encoded input, which would increase the memory consumption (larger input matrix on the ESL) and would require more samples to abstract a pattern. To speed up the learning process we reduced the input space to only four symbols, those strictly necessary to identify a correctly formatted email address. Table 5.2.1 shows the map used to create the 4-symbols dataset. The reduced input vector implies faster vector-matrix multiplication (low dimensionality) and less memory consumption (due to a smaller matrix size). In addition, the reduced input space improves the learning speed.

ASCII Domain	4-Symbols Domain	
Original Symbols	Target Symbol	Target Symbol Index
[\t\n\r]+	Single space	0
[a–zA–Z0–9]	x	1
@	@	2
[.]+	Single dot	3

Table 5.2.1: Map from ASCII domain to 4-symbols

For the comparison we configured the different algorithms (FIMT-DD, feed-forward NN, ESN and ESHT) as shown in Table 5.2.2. For the ESN we explored different values for the α, *density* and *learning rate* hyper-parameters in the range $[0.1, 1.0]$ and linear output. For the standard NN we also explored values for the *learning rate* in the same range. In order to configure ESHT, we used the results obtained for *Counter Opt2* in section 5.2.2, with $\alpha = 1.0$ and *density*=0.4. We increased the number of neurons to 4,000.

Algorithm	Density	α	Learning rate	Loss	# Errors	Accuracy (%)
FIMT-DD	-	-	-	4,119.7	336	91.61
NN	-	-	0.8	2,760	88	97.80
ESN1	0.2	1.0	0.1	1,032	57	98.47
ESN2	0.7	1.0	0.1	850	61	98.47
ESHT	0.1	1.0	-	**180**	**10**	**99.75**

Table 5.2.2: Email address detector results

The first conclusion from the results shown in Table 5.2.2 is the well known inability of both FIMT-DD and NN to capture time dependencies in the input. The NN defaults to the majority class (always predicts a *0* symbol), achieving 97.80% of accuracy (88 errors, the total number of correct email addresses in the dataset input) with loss of 2,760 (the length of all emails in the dataset). The FIMT-DD obtains the worst accuracy (91.61% with loss 4,119.70).

ESHT clearly outperforms the two best configurations obtained for ESN, with only 10 errors and a cumulative loss of 180 (compared to around 60 errors and cumulative error around 1,000 in ESN). In order to better understand the results shown in Table 5.2.2 for ESN and ESHT, the top plot in Figure 5.2.10 shows the *cumulative loss* evolution with the number of samples in the input. After eight repetitions the ESN failed to get right all the 11 emails in the same block (observe how the *cumulative loss* continues to grow with the number of samples). The ESHT clearly outperforms the ESN with only 500 samples; after this number of samples, the plot shows a constant loss for the ESHT between 500 and 1,000 samples (this is an effect of the plot scale, in this range the loss grows, but after this it stays constant).

The bottom plot in Figure 5.2.10 shows the evolution of the accuracy of ESN and ESHT with the number of samples in the input. Observe that the three curves start with 100% accuracy; this is due to the fact that the first label is zero, and all tests started biased to zero.

5.2.3 Classification evaluation methodology and real-world datasets

The classification version of ESHT is evaluated using real-world datasets widely used for data streams classification. The datasets used were COVT and ELEC. The SUSY dataset used in Chapter 4 has not been used due to its excessive execution time (+24hrs, where MOA only needed few minutes to finish in the worst case). Our proof of concept implementation did not used GPUs for accelerating computations required by ESL.

The temporal capabilities of our proposed design were tested using a single HT as a reference. Also, we compare ESHT to the best results obtained in Section 4.2.2, which were obtained using two state-of-the-art methods: Leveraging Bagging (LB) and k-nearest neighbours (kNN).

The evaluation scheme used in this evaluation is the prequential evaluation approach typically used in datastreams (see Section 3.3).

5.2.4 Classification evaluation

As in the regression evaluation, combinations of α and density in the range $[0.1, 1.0]$ in steps of 0.1, and neurons in the hidden layer from 10 to 100 (with increments of 10) were tested in order to assess their impact in the final accuracy.

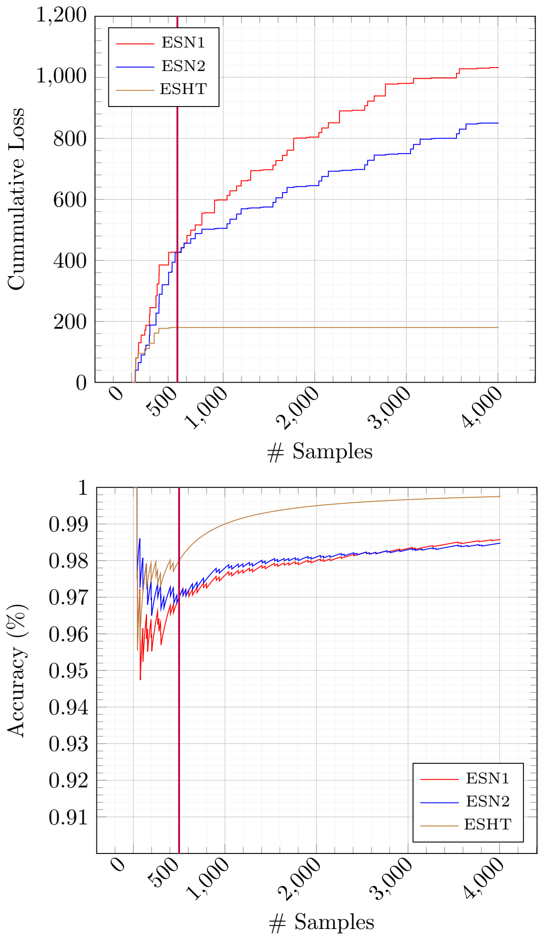


Figure 5.2.10: *Cumulative loss* (top) and *accuracy* (bottom) evolution for *emailFilter*

Table 5.2.3 summarizes the best results obtained by ESHT and its configuration. ESHT improves a single HT in both datasets, which was our initial target. In addition, ESHT outperformed LB, an ensemble of HTs, and kNN by almost 4 percentage points in the COVT dataset. In the ELEC dataset, LB is still providing the best accuracy, outperforming ESHT by 8.5 points. Regarding the hyper-parameters configuration, observe that the only difference in configuration was in the α hyper-parameter, the other two hyper-parameters had the same values for density (1.0) and number of neurons (10).

A detailed influence in the accuracy of the α hyper-parameters for the best results configuration (10 neurons and density 1.0) is shown for each dataset in Figure 5.2.11. Despite both plots having a clear peak where best accuracy was achieved, in some cases results vary in an unpredictable way when increasing α by 0.1. For example, in the ELEC dataset, using $\alpha = 0.6$ leads to the worst

	ESHT Best Results					HT		State-of-the-art		
Dataset	Acc (%)	rank	α	Density	neurons	Acc (%)	rank	Acc (%)	rank	Alg
ELEC	81.3	1	0.9	1.0	10	79.2	3	**89.8**	1	LB
COVT	**96.1**	1	0.2	1.0	10	80.3	3	92.2	2	kNN

Table 5.2.3: ESHT performance comparison against a single HT, and the best results obtained on each dataset.

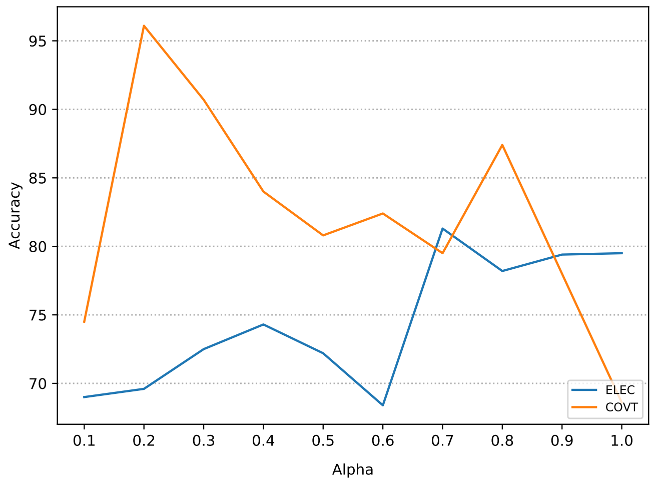


Figure 5.2.11: Influence of α on the ESHT accuracy; Fixed density=1.0 and 10 neurons in the ESL

	State-of-the-Art		Hoeffding Tree	ESHT
Dataset	Method	Time (s)	Time (s)	Time(s)
ELEC	LB	10	1	50.2
COVT	kNN	605	19	2434
SUSY	LB	530	45	-

Table 5.2.4: Comparing ESHT execution time against other data-streams methods.

accuracy, but increasing α by 0.1 ($\alpha = 0.7$) yields to the best accuracy. A similar observation can be done in the COVT dataset for $\alpha = 0.1$ and $\alpha = 0.2$.

Unfortunately, as shown in Table 5.2.4, ESHT major issue is the computation time which is about one order of magnitude higher than other popular data streaming methods. Most of the extra time needed to process the input data is due to ESL since all matrix multiplications were computed in the CPU.

5.3 Summary

This chapter proposes the Echo State Hoeffding Tree (ESHT) for learning temporal dependencies in data streams in real time. The proposal is based on the combination of the reservoir in the Echo State Network (ESN) and a Hoeffding Tree (on classification problems) or a FIMT-DD (on regression problems).

The ESHT regression version was evaluated with a proof-of-concept implementation and three string-based input sequences generated by functions typically implemented by a programmer, opening the door to explore the possibilities of Learning Functions instead of programming them. ESHT was able to learn faster than standard ESN and requires fewer hyper-parameters to be tuned (only two). In addition, these two hyper-parameters have a more predictable effect on the final accuracy than typical neural networks hyper-parameters such as learning rate or momentum.

On classification problems, our proof-of-concept implementation was tested on some of the datasets used in Chapter 4. The results show that ESHT can capture temporal dependences, outperforming a single HT significantly (accuracy difference is larger then one percentage point). Compared to LB (an ensemble of HT), ESHT could only outperform it in one out of two datasets. The limiting factor was the execution time which limits the number of samples throughput, and being the main reason why SUSY dataset evaluation was aborted. Using hardware accelerators such as GPU may reduce significantly the ESL computation time. However, we still need to face the problem that the final ESHT accuracy is not as good as LB for some datasets.

Given the issues we are facing with NNs for data streams, in the next two contributions, we decided to focus on ensembles exclusively. This way, in Chapter 6 we reduce the number of learners used in ensembles, enabling their deployment on more constrained hardware environments. Later, in Chapter 7, we propose a HT and an ensemble design that fully exploits modern CPUs capabilities in order to reduce the response time down to a few microseconds.

6 Resource-aware Elastic Swap Random Forest for Evolving Data Streams

Leveraging Bagging (LB), an ensemble of Hoeffding Trees, obtained consistent results (usually the best) in the evaluations of Chapters 4 and 5. This is not a coincidence, since ensemble learners are the preferred choice for processing evolving data streams due to their better classification performance over single models. LB was considered the state-of-the-art classifier in MOA [12] at the moment of starting this dissertation. However, the ADAPTIVE RANDOM FOREST (*ARF*) [44] was introduced in 2017 and since then, it is currently considered the state-of-the-art ensemble for classifying evolving data streams in the MOA.

The number of learners in the ensemble is usually decided independently of the characteristics of the data stream. For example, the *ARF* uses by default 100 random decision trees, or the LB in Chapters 4 and 5 used 10 decision trees. In this chapter, we argue that this number of learners in *ARF* is not optimal and that we can get similar results using a smaller number of learners. For this, we introduce a new adaptive methodology to automatically decide the number of learners to be used in incremental models.

In this contribution, the originally proposed *ARF* is extended in two orthogonal directions. On one side, ELASTIC SWAP RANDOM FOREST (*ESRF*) splits the learners in two groups: a forefront group that contains only the learners used to do predictions, and a second candidate group that contains learners trained in the background and not used to make predictions. At any time, a learner in the forefront group can be replaced by a candidate learner if this swapping operation improves the overall ensemble accuracy. On the other side, *ESRF* extends *ARF* by dynamically increasing/decreasing the number of learners in the ensemble, in particular the number of learners in the forefront set.

6.1 Preliminaries

By default, the reference implementation for *ARF* in the *Massive Online Analysis (MOA)* framework uses 100 random trees (ARF100). While this configuration provides competitive accuracy we have observed that accuracy can converge for sizes lower than 100 random trees, which is the case for almost all datasets used in this chapter as shown in Figure 6.1.1. The convergence point is different for each dataset, but once the accuracy converges, adding extra trees only increases computational and memory costs with marginal impact in the accuracy. For example, the RBF_f dataset requires 70 learners in the ensemble to achieve a percentage difference in terms of accuracy (with respect to ARF100) in the second decimal place; however, the COVT dataset achieves the same difference in accuracy with only 30 learners.

Table 6.1.1 details ARF results for sizes 40, 50, 100, and 200 (namely ARF40, ARF50, ARF100 and ARF200 respetively). For each dataset, results are expressed in terms of accuracy difference with respect the best result. *ARF* requires at least 50 learners for all results to be within the one point range

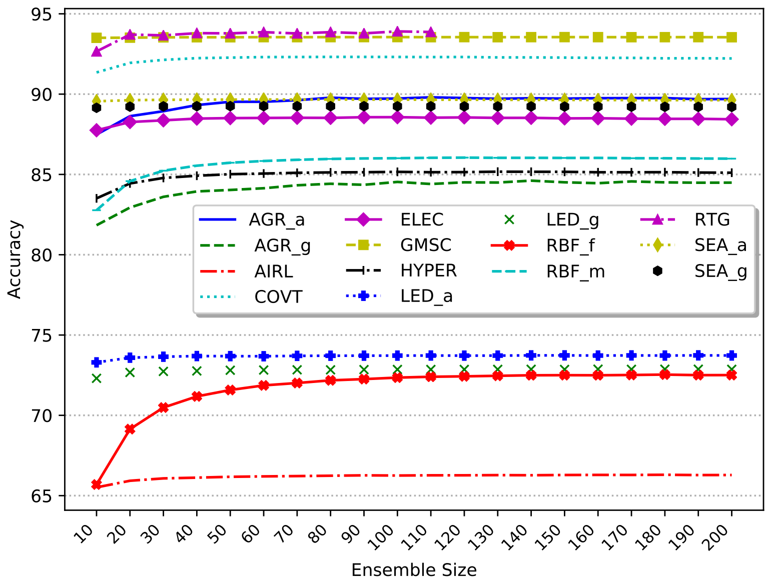


Figure 6.1.1: ARF accuracy evolution with ensemble size.

Table 6.1.1: Difference in accuracy for difference *ARF* sizes with respect the best one. A negative result means worse then the best.

	ARF Best		$\Delta_{Accuracy}$ w.r.t. ARF Best			
Dataset	Acc(%)	Size	ARF40	ARF50	ARF100	ARF200
AGR_a	89.80	110	-0.48	-0.28	-0.07	-0.10
AGR_g	84.61	140	-0.67	-0.58	-0.08	-0.12
HYPER	85.17	130	-0.25	-0.16	-0.01	-0.06
LED_a	73.73	190	-0.04	-0.04	-0.01	-0.004
LED_g	72.87	160	-0.11	-0.06	-0.01	-0.01
RBF_f	72.54	180	-1.36	-0.96	-0.19	-0.03
RBF_m	86.05	120	-0.50	-0.32	-0.04	-0.07
RTG	93.91	100	-0.11	-0.13	0	-
SEA_a	89.66	70	-0.013	-0.009	-0.01	-0.048
SEA_g	89.25	60	-0.002	-0.002	-0.01	-0.05
AIRL	66.30	180	-0.18	-0.13	-0.05	-0.02
COVT	92.32	80	-0.08	-0.05	-0.01	-0.10
ELEC	88.57	100	-0.09	-0.06	-0.005	-0.13
GMSC	93.55	100	-0.01	-0.01	-0.002	-0.01
Average	84.17	122.86	-0.28	-0.20	-0.04	-0.05

(i.e., the difference in accuracy is larger than one percentage point). Observe that ARF40 performed significantly worse in the RBF_f evaluation since the difference is above one percentage point. Is between 50 and 100 learners where the accuracy curve converges for all datasets (it can also be seen in Figure 6.1.1): observe the marginal differences in accuracy for ARF100 with respect the best results. Further from 100 learners, increasing the ensemble size from 100 to 200 has marginal impact in the accuracy. However, memory and CPU requirements are doubled since they grow with the number of learners in the ensemble.

In terms of memory, in the worst case scenario *ARF* allocates one background learner for each active learner in the ensemble, requiring twice the memory. However, background learners are simpler since they do not need to keep any drift detection data structure. In practice, the number of background learners is usually lower than the number of active learners as it can be appreciated in Table 6.1.2, being 5 and 10 the average number of background learners used by *ARF50* and *ARF100* respectively.

Regarding the execution time, each active and background tree needs to be traversed for each new arriving instance. With the aim of reducing the computational and memory requirements of the ensemble, enabling either high–throughput implementations or their deployment in resource–constrained devices, this chapter proposes the ELASTIC SWAP RANDOM FOREST algorithm described in the next section.

Table 6.1.2: Average number of background learners for ARF with 100 and 50 learners. Note that RTG dataset has no drift, thus, ARF needed 0 background learners

	ARF_100			ARF_50		
Dataset	Mean	stdev	max	Mean	stdev	max
AGR_a	5.25	3.67	39.00	2.74	2.47	21.00
AGR_g	8.75	4.02	40.00	4.40	2.51	20.00
HYPER	9.50	7.95	39.00	4.26	3.83	19.00
LED_a	10.04	8.55	51.00	4.74	3.63	26.00
LED_g	8.07	4.05	37.00	4.62	2.54	24.00
RBF_f	20.55	4.44	41.00	10.26	3.08	24.00
RBF_m	13.60	4.28	31.00	6.74	2.75	20.00
RTG	0.00	0.00	0.00	0.00	0.00	0.00
SEA_a	0.06	1.41	54.00	0.03	0.68	31.00
SEA_g	0.68	4.09	53.00	0.31	2.02	26.00
airlines	7.95	4.81	31.00	3.95	2.72	19.00
covtypeNorm	38.01	12.20	70.00	19.07	6.50	37.00
elecNormNew	28.82	9.62	55.00	13.29	5.29	27.00
GMSC	0.65	0.81	2.00	0.49	0.72	3.00
Average	10.13	4.66	36.20	4.99	2.58	19.80

6.2 Elastic Swap Random Forest

Elastic Swap Random Forest (*ESRF*) is a fast streaming random forest based method that adapts its size in an elastic way, to be consistent with the current distribution of the data. *ESRF* also includes a swap component that maintains two pools of classifiers to decide which ones are actually used for better prediction making. Although the elastic and swap components in *ESRF* are independent, they are presented together in Algorithm 3.

The swap component in *ESRF* divides the classifiers into two groups: the foreground (or active) learners and the background (or candidate) learners. The *Forefront Set* (FS) contains those classifiers with higher accuracy that are used for predicting; the *Candidates Set* (CS) contains those classifiers that are trained but nor used for prediction since they accuracy is low compared to those on the FS. For each arriving instance X, the prediction is done just using the learners in FS (lines 14–16). During training (lines 1–6), as done in *ARF*, all classifiers are trained simulating bagging by weighting the instance according to a $Poisson(\lambda = 6)$. After that (lines 4–5), *ESRF* swaps the worst classifier in FS (i.e. the one with the lowest accuracy f_{min}) with the best in CS (i.e. the one with the highest accuracy c_{max}) when the later becomes more accurate. With the swap component the number of learners required in the ensemble is $|FS| + |CS|$; all the learners in these two sets are trained for each arriving instance.

The elastic component in *ESRF* dynamically determines the size of the FS. This elastic component is not tied to *ESRF* and it could be implemented in any ensemble method such as the original *ARF*. For each arriving instance, the algorithm checks if FS needs to be resized (line 2), adding or removing r new learners. In case of growing, the new r learners could be taken from the CS set; however, and in order to add more diversity in the ensemble, the elastic component introduces a third set of learners, the *Grown Set* (GS) with r learners, which also nees to be trained on each arriving instance and is reset on every resizing operation.

Algorithm 4 details how the elastic component in *ESRF* works. For each

Algorithm 3 ELASTIC SWAP RANDOM FOREST

```
Require:
    $X$: input sample
    $z$: label for input sample
    $FS$: Set of forefront learners
    $CS$: Set of candidate learners
    $r$: Resize factor
    $GS$: Set of $r$ random trees
    $f_{min}$: Classifier from $FS$ with lowest accuracy
    $c_{max}$: Classifier from $CS$ with highest accuracy
 1: function TRAINONINSTANCE($X$)
 2:     RESIZEENSEMBLE($X$, $z$)
 3:     TRAINALLCLASSIFIERS($X$)
 4:     find $c_{max}$ and $f_{min}$
 5:     swap classifiers if $c_{max}$ is more accurate than $f_{min}$
 6: end function
 7: function TRAINALLCLASSIFIERS($x$)
 8:     for each classifier $c \in FS \cup CS \cup GS$ do
 9:         $w \leftarrow$ Poisson($\lambda = 6$)
10:         Set the weight of $x$ to $w$
11:         Train classifier $c$ using $x$
12:     end for
13: end function
14: function PREDICTONINSTANCE($X$)
15:     return PREDICTLABEL($X$, $FS$)
16: end function
17: function PREDICTLABEL($x$, $S$)
18:     for each classifier $c \in S$ do
19:         $\hat{y_c} \leftarrow$ prediction for $x$ using $c$
20:     end for
21:     $\hat{y} \leftarrow$ combination of predictions $\hat{y_c}$
22:     return $\hat{y}$
23: end function
```

arriving instance, *ESRF* simulates having three ensembles of random trees (lines 2–4):

- the default one (i.e., the current ensemble with $|FS|$ learners in FS);
- the shrunk ensemble containing only the $|FS| - r$ learners with higher accuracies in FS;
- and the grown ensemble containing the learners in FS FS plus the r extra learners in the Grow set GS (i.e. in total $|FS| + r$ learners).

Line 5 decides whether resizing is needed, based on the performance of each of these three ensembles. *ESRF* may decide to keep the current configuration for the ensemble or to apply a resize operation if either the shrunk or grown ensemble improve the default ensemble performance. In this case, the r learners in GS are added to FS in case growing is decided (lines 6–9) or the r classifiers in FS with lowest accuracy are removed in case shrinking is decided (lines 10–13). Observe that the proposed elastic component does not use the candidates set CS, although this is an alternative option that is not considered due to the limit in the number of pages.

The performance for an ensemble is computed using the exponential weighted moving average (EWMA) of its accuracy. EWMA gives larger weight to recent data, and a smaller weight to the older one. The weighting factor decreases exponentially but never reaches zero. It is calculated using this formula:

$$\text{EWMA}_i = \text{EWMA}_{i-1} + \alpha * (S_i - \text{EWMA}_{i-1})$$

where S_i is the current value being added, α is the weighting factor defined as $\alpha = exp(1/W)$, where W is a fixed time window. In *ESRF*, $W = 2000$ and $S_i \in \{0, 1\}$, 1 for a label predicted correctly and 0 otherwise. EWMA allows *ESRF* to keep track of each ensemble accuracy without being influenced too much by past prediction results.

The necessary logics to decide whether the ensemble should be resized or not are detailed in function CheckIfResize in Algorithm 4. First, it updates each ensemble EWMA (lines 16–18); then compares the EWMA estimation of the default ensemble with the other two ensembles to compute the differences Δ_{shrink} and Δ_{grown} (line 19–20). If Δ_{grown} is larger than Δ_{shrink}, and Δ_{grown} is above a threshold, then a grow operation is triggered (lines 21–23). The shrink operation is decided in lines 24–26 and works similarly. In case $\Delta_{grown} = \Delta_{shrink}$, *ESRF* favours growing against shrinking by comparing Δ_{grown} first.

As mentioned above, only classifiers in FS are used to make the ensemble prediction (lines 14–16 in Algorithm 3). *ESRF* implements the same weighting voting policy for instances as in *ARF*: each classifier has an associated weight that is computed as the number of correctly classified instances divided by

Algorithm 4 Check resize and update size for *ESRF*

```
Require:
    $r$: Resize factor
    $FS$: Set of forefront learners
    $FS_{min}$: Set of $r$ learners from $FS$ with lower accuracy
    $GS$: Set of $r$ random trees
    $SRK$: Shrunk ensemble $FS \setminus FS_{min}$
    $GRN$: Grown ensemble $FS \cup GS$
    $T_g$: grow threshold
    $T_s$: shrink threshold
 1: function ResizeEnsemble($x$, $z$)
 2:     $\hat{y_s} \leftarrow$ PredictLabel($x$, $SRK$)
 3:     $\hat{y_d} \leftarrow$ PredictLabel($x$, $FS$)
 4:     $\hat{y_g} \leftarrow$ PredictLabel($x$, $GRN$)
 5:     Operation $\leftarrow$ CheckIfResize($z$, $\hat{y_s}$, $\hat{y_d}$, $\hat{y_g}$)
 6:     if Operation==GROW then
 7:         $FS = FS \cup GS$
 8:         Start new $GS$ with $r$ new trees
 9:     end if
10:     if Operation==SHRINK then
11:         $FS = FS \setminus FS_{min}$
12:         Start new $GS$ with $r$ new trees
13:     end if
14: end function
15: function CheckIfResize($z$, $\hat{y_s}$, $\hat{y_d}$, $\hat{y_g}$)
16:     Update EWMA$_{shrunk}$ using $\hat{y_s}$ and $z$
17:     Update EWMA$_{default}$ using $\hat{y_d}$ and $z$
18:     Update EWMA$_{grow}$ using $\hat{y_g}$ and $z$
19:     $\Delta_{shrink}$ = EWMA$_{shrunk}$ − EWMA$_{default}$
20:     $\Delta_{grow}$ = EWMA$_{grown}$ − EWMA$_{default}$
21:     if $\Delta_{grow} > \Delta_{shrink}$ and $\Delta_{grow} > T_g$ then
22:         return GROW;
23:     end if
24:     if $\Delta_{shrink} > \Delta_{grow}$ and $\Delta_{shrink} > T_s$ then
25:         return SHRINK
26:     end if
27: end function
```

the total number of instances since last reset (due to concept drift), reflecting the classifier performance on the current concept. To cope with evolving data streams, a drift detection algorithm is used with each learner of the ensemble algorithm described above (not shown in algorithm 3). *ESRF* resets a tree as soon as it detects concept drifting. This is much simpler than the drift detection algorithm in *ARF* since there is a single threshold and no background learners are created when drift is detected.

6.3 Experimental Evaluation

ELASTIC SWAP RANDOM FOREST (*ESRF*) has been implemented using MOA (Massive Online Analysis) framework, and evaluated using datasets shown in Table 6.3.1 (see Chapter 3 for more details). We gradually evaluate the impact of the two components in the proposed *ESRF*. First we evaluate the performance of the swap component and then we evaluate the impact of adding the elastic component, always comparing against the baseline *ARF* ensemble with 100 learners (ARF100). Two parameters in *ESRF* are fixed in the evaluation presented in this chapter: $|CS| = 10$ and $r = |GS| = 1$. Regarding the number of learners in *CS*, we have observed that it does not have a major impact in terms of accuracy for values larger than 10; in order to make a fair comparison in terms of resources when comparing with *ARF*, we set the value to 10, which coincides with the average number of background learners that are required by the drift mechanism in *ARF* for the datasets used in this chapter. Regarding the resize factor in the elastic component, we have evaluated values of 1 and 5.

The parameters that are evaluated are: $|FS|$ (with a minimum value of 15 and maximum limited to $|FS| + |CS| + |GS| = 100$) and the two thresholds that decide resizing (T_g and T_s). The rest of hyper-parameters, which are common for both ESRF and ARF100, are set to their default values in MOA.

For completeness, later in this section we evaluate the performance of only using the elastic component in the *ESRF*. We will refer to this configuration as

Dataset	Samples	Attrs	Labels	Generator	Drift
AGR_a	1,000,000	9	2	Agrawal	A
AGR_g	1,000,000	9	2	Agrawal	G
HYPER	1,000,000	10	2	Hyperplane	I.F
LED_a	1,000,000	24	10	LED Drift	A
LED_g	1,000,000	24	10	LED Drift	G
RBF_m	1,000,000	10	5	RBF	I.M
RBF_f	1,000,000	10	5	RBF	I.F
RTG	1,000,000	10	2	RTG	N
SEA_a	1,000,000	3	2	SEA	A
SEA_g	1,000,000	3	2	SEA	G
AIRL	539,383	7	2	-	-
COVT	581,012	54	7	-	-
ELEC	45,312	8	2	-	-
GMSC	150,000	11	2	-	-

Table 6.3.1: Synthetic (top) and real-world (bottom) datasets used for performance evaluation and comparison. Synthetic datasets drift type: A (abrupt), G (gradual) I.F (incremental fast), I.M (incremental moderate), N (None)

Elastic Random Forest (ERF). Other than not using the swap component, main difference with respect *ESRF* of this configuration is that it uses the same drift detection used in *ARF*.

The hardware platform that has been used to conduct the performance analysis is an Intel(R) Xeon(R) Platinum 8160 CPU running at 2.10GHz (24 cores, 48 threads), 96GB of RAM, SUSE Linux Enterprise Server 12 SP2 (kernel 4.4.120-92.70-default) and openJDK 64bits 1.8.0_161.

The evaluation methodology that has been used to conduct all the experiments reported is streaming prequential CV detailed in Section 3.3.2.

6.3.1 Swap Random Forest

This subsection individually evaluates the swap component in *ESRF* in terms of accuracy and ensemble size, using ARF100 as a reference. We will refer to this *ESRF* ensemble configuration as Swap Random Forest (*SRF*).

Figure 6.3.1 shows the accuracy obtained by *SRF* when using a fixed number of learners in the FS. Comparing to Figure 6.1.1 one can appreciate that the accuracy of *SRF* converges faster than ARF100. With only 35 learners in FS the differences in accuracy are less than 1 percentage point, while ARF100 required 50 learners to be within the same range (see Table 6.1.1).

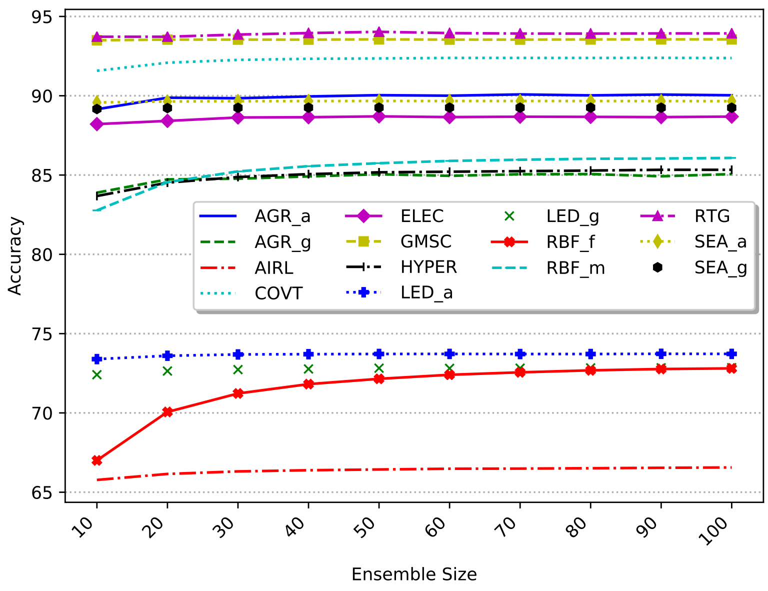


Figure 6.3.1: SRF accuracy evolution as the ensemble size increases

Table 6.3.2 details the results for two *SRF* configurations (SRF F35 and F50) and ARF100. The Δ_{acc} column always shows the difference with ARF100. When using 35 learners in the front set of *SRF*, the differences observed in terms of accuracy are marginal in most of the tests, except for the two RBF datasets that are a little bit more noticeable; however, in terms of resources, *SRF* is only using one third of the trees that are used in ARF100. When using 50 learners, half of the trees compared to ARF100, *SRF* outperforms ARF100 in 11 of the 14 datasets, being the differences in the other 3 smaller than before. The reduction in size for F50 implies a speedup of 2.03x on average. Looking at the average for all datasets, we conclude that the average difference is very small (0.07 worse for F35 and 0.06 better for F50 wrt ARF100).

The average standard deviation around the mean accuracy (of 10 runs) are the same for ARF100 and SRF50, while SRF35 is slightly larger than ARF100 (see stdev columns in Table 6.3.2). While SRF50 is able to reduce stdev in some datasets, SRF35 has larger variance in almost all datasets. In general, the lower the number of learners the harder it is to compensate miss classified outputs, which causes more fluctuations in the combined voting. However, the absolute value of Δ_{acc} between ARF100 and SRF50 is never larger than their respective stdev columns, and similarly for ARF100 and SRF35. Therefore, we can conclude that there is no significant difference between them in any of the datasets tested. This was expected since both algorithms use the same set of F learners that are contributing more to the combined voting (those with higher accuracy). In fact, at any given time step SRF is equivalent to an ARF that places zero weighting to those learners not in the SRF forefront set.

Table 6.3.2: Accuracy comparison between ARF100 and SRF with $|FS| = 35$ and $|FS| = 50$

	ARF100		SRF F35			SRF F50		
Dataset	Acc(%)	Stdev	Acc(%)	Stdev	Δ_{acc}	Acc(%)	Stdev	Δ_{acc}
AGR_a	89.73	0.07	**90.13**	0.11	0.40	90.04	0.08	0.31
AGR_g	84.53	0.01	84.81	0.08	0.28	**85.05**	0.08	0.52
HYPER	85.16	0.01	84.96	0.02	-0.20	**85.17**	0.01	0.01
LED_a	**73.72**	0.02	73.69	0.02	-0.03	**73.72**	0.01	0.00
LED_g	**72.86**	0.01	72.77	0.01	-0.09	72.82	0.00	-0.04
RBF_f	**72.35**	0.01	71.59	0.01	-0.76	72.15	0.01	-0.20
RBF_m	**86.01**	0.00	85.41	0.01	-0.60	85.75	0.01	-0.26
RTG	93.91	0.20	93.82	0.19	-0.09	**94.03**	0.20	0.12
SEA_a	89.66	0.00	89.66	0.00	0.00	**89.67**	0.00	0.01
SEA_g	89.24	0.00	**89.25**	0.00	0.01	**89.25**	0.00	0.01
AIRL	66.25	0.01	66.34	0.01	0.09	**66.43**	0.01	0.18
COVT	92.31	0.01	92.30	0.01	-0.01	**92.35**	0.01	0.04
ELEC	88.57	0.02	88.62	0.03	0.05	**88.70**	0.05	0.13
GMSC	**93.55**	0.01	93.54	0.00	-0.01	**93.55**	0.00	0.00
Average	84.13	0.03	84.06	0.04	-0.07	84.19	0.03	0.06

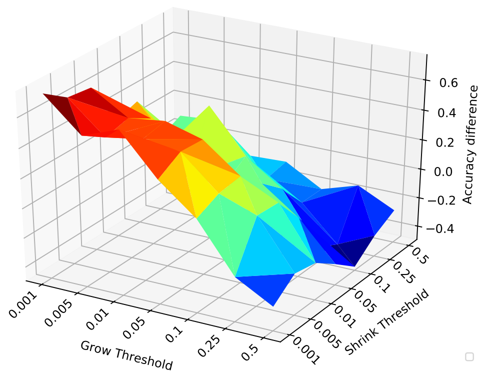
6.3.2 Elastic Swap Random Forest

In this subsection we evaluate the complete *ESRF* ensemble, using both the swap and elastic component together. Once we have seen in the previous subsection how the swap component is able to significantly reduce the number of learners required in the ensemble, we want to see if the elastic component is able to dynamically determine the most appropriate size for the FS. As before, accuracy and ensemble size are compared against the ARF100 baseline.

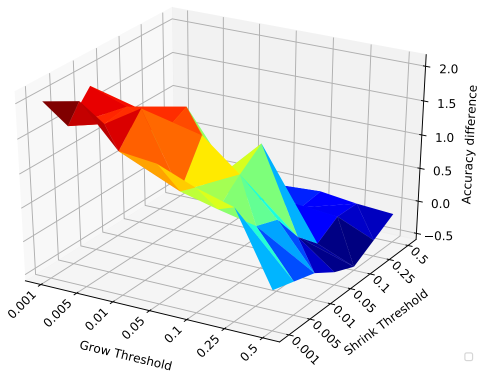
In order to asses the impact of grow and shrink thresholds (T_g and T_s in Algorithm 4, respectively) in the accuracy and ensemble size, different sensitivity levels have been tested. Restrictive thresholds allow a resize operation only when the difference for Δ_{shrunk} or Δ_{grown} is in first decimal or higher; similarly, permissive thresholds allow differences in the second or third decimal. In general, more permissive thresholds make *ESRF* grow faster, which in time, reduces the differences with ARF100; however more restrictive thresholds have the opposite effect. Consequently, a generic strategy for improving the accuracy is to use of $T_g = T_s$ and move along the diagonal from more restrictive thresholds ($T_g = T_s = 0.5$) to more permissive ones ($T_g = T_s = 0.001$), as it can be appreciated in Figure 6.3.3 (real-world datasets) and Figure 6.3.2 (synthetic datasets). Although this generic strategy yields to an improvement in the accuracy for all cases, observe that datasets such as SEA_a, SEA_g, LED_g and ELEC, obtained the best accuracy using a combination of restrictive shrink and permissive grow thresholds.

The two thresholds can also be set to target different target scenarios. For example, on hardware platforms, such as ARM boards typically used in Internet of Things (IoT) applications, it may be desirable to make the ensemble only grow if strictly needed and force it to reduce its size as soon as it can in order to save memory and computation time. This can be achieved by using a shrink threshold (T_s) lower than the grow threshold (T_g), for example $T_g = 0.01$ and $T_s = 0.001$. Table 6.3.3 details results obtained with this configuration. Observe that the average performance of *ESRF* and *ARF100* are very similar (*ESRF* worse with -0.08 difference). In 11 out of the 14 datasets, the difference in accuracy is never worse than -0.23, being able to outperform *ARF100* by 1 percentage point in the AGR_g dataset. The three datasets presenting more challenges to this configuration are RTG, RBF_f, RBF_m, being the last the worst performing one (-0.81 difference). As shown in the central and right part of this table, the main advantage of this configuration is that the average number of trees used in the ensemble is 22, (4.5 times less trees than *ARF100*, with the same proportional reduction in memory used); in 10 out of the 14 datasets *ESRF* never required more than 42 trees. This reduction in the ensemble size implies an average speedup of 3.84x, or in other words, may allow the use of devices up of 3.84x less powerful.

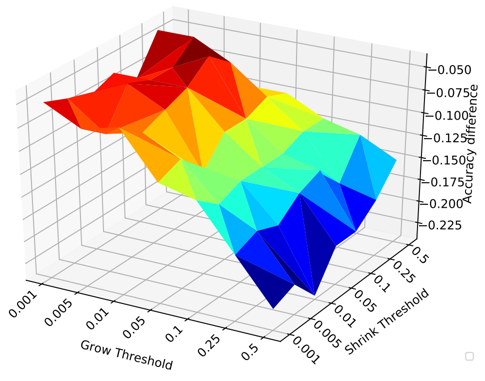
For other scenarios requiring higher accuracy, or in which memory or CPU time are not an issue, *ESRF* may use more sensitive thresholds in order to grow faster and reach higher accuracy requirements. For example, Table 6.3.4 shows the results that are obtained when using $T_g = T_s = 0.001$. This configuration allows ESRF to grow larger in those datasets that were more challenging in Table 6.3.3, while keeping similar average size for the rest of the



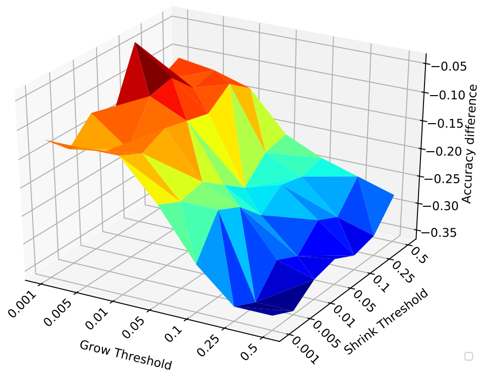
(a) AGR_a



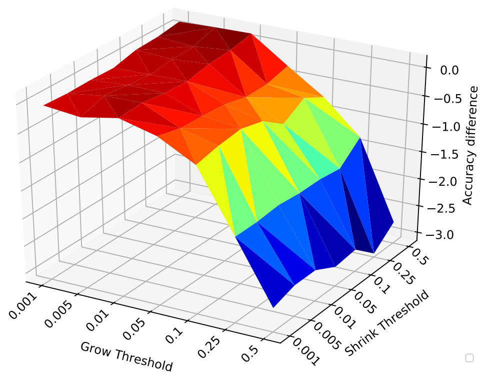
(b) AGR_g



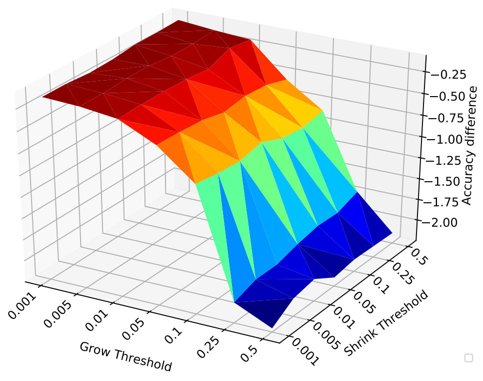
(c) LED_a



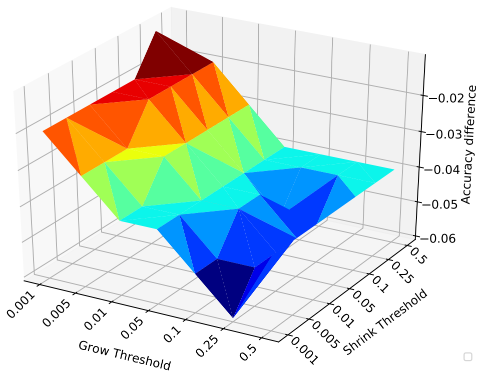
(d) LED_g



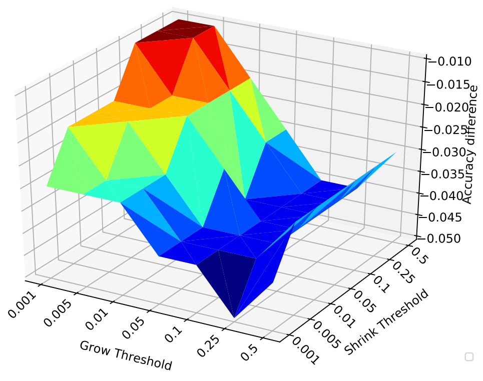
(e) RBF_f



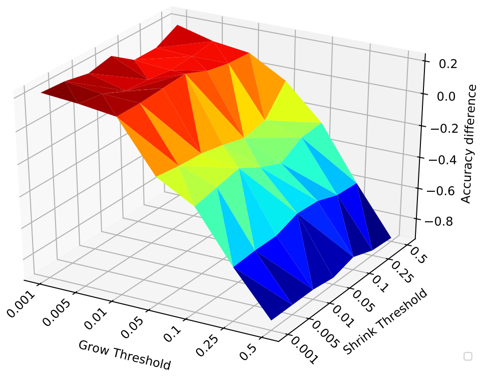
(f) RBF_m



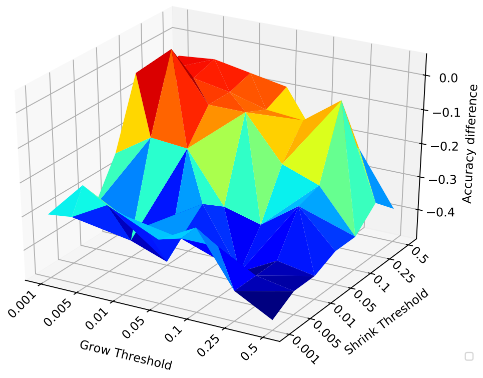
(g) SEA_a



(h) SEA_g



(i) HYPER

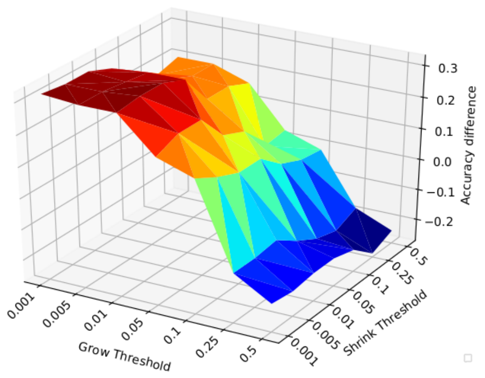


(j) RTG

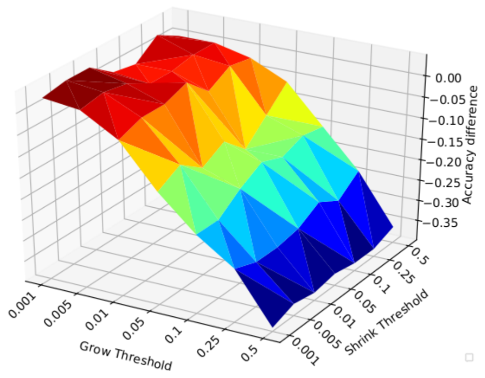
Figure 6.3.2: Influence of using a resize factor $r = 1$ in the accuracy distance w.r.t. ARF100 of the grow and shrink thresholds on the synthetic datasets. Negative numbers means ARF100 is better.

datasets. This narrows the difference in accuracy to be not lower than -0.05 in 12 out of the 14 datasets, and in the worst case -0.3 (RTG). In terms of number of learners and speedup the same table shows a reduction of the execution time by 3.14x and the use of 33 learners on average, always compared to ARF100.

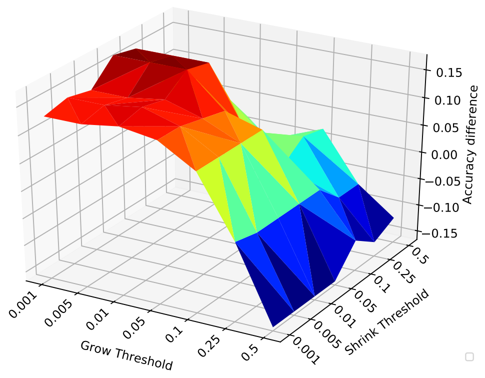
For completeness, the results for *ESRF* using a resize factor $r = 5$ are detailed in Figure 6.3.5 (real-world datasets) and Figure 6.3.4 (synthetic datasets). In this configuration, the generic strategy mentioned above for $r = 1$ (moving in the diagonal $T_g = T_s$) does not work as expected. For example, moving to more permissive thresholds in the RBF_f dataset, results in accuracy differences larger than one percentage point. A generic observation



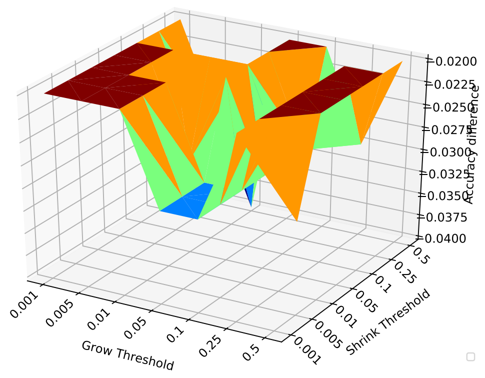
(a) AIRL



(b) COVT



(c) ELEC



(d) GMSC

Figure 6.3.3: Influence of using a resize factor $r = 1$ in the accuracy distance w.r.t. ARF100 of the grow and shrink thresholds on the real-world datasets. Negative numbers means ARF100 is better.

is that permissive shrink thresholds affects negatively the *ESRF* accuracy in most datasets.

A proper strategy for $r = 5$ is to fix the shrink threshold to $T_s = 0.5$ and only varying the grow threshold (T_g). Table 6.3.5 details the case for $T_g = 0.01$ and $T_s = 0.5$, which provides a similar accuracy than *ARF100* using an average of 50 learners. Larger resize factors makes the *ESRF* grow faster, which in time, increases the ensemble performance at expenses of consuming more CPU and memory due to the extra learners required.

In our opinion, a resize factor $r = 1$ the best option since it provides a good balance between size and accuracy. Also, the strategy for balancing this trade-off is very intuitive: only requires moving in the diagonal $T_g = T_s$, where, more permissive the thresholds yield better performance at expenses of increasing the ensemble size.

Table 6.3.3: ESRF comparison with ARF100. Resource-constrained scenario: $T_g = 0.01$ and $T_s = 0.001$

	ARF100		ESRF accuracy			ESRF speedup		ESRF size			
Dataset	Acc(%)	Time(s)	Acc(%)	Rank	Delta	Time(s)	Speedup	mean	stdev	max	min
AGR_a	89.73	22670.86	**89.96**	1	0.23	4423.66	5.12	11.86	2.49	25.00	10.00
AGR_g	84.53	24630.29	**85.55**	1	1.02	4935.70	4.99	12.49	2.76	28.00	10.00
HYPER	**85.16**	20157.96	84.95	2	-0.21	7506.14	2.69	32.96	8.07	50.00	10.00
LED_a	**73.72**	15344.18	73.59	2	-0.13	3439.01	4.46	19.74	6.32	38.00	10.00
LED_g	**72.86**	15319.00	72.63	2	-0.23	3439.02	4.45	19.39	6.35	40.00	10.00
RBF_f	**72.35**	19938.81	71.97	2	-0.38	9385.82	2.12	44.88	9.04	57.00	10.00
RBF_m	**86.01**	19638.30	85.20	2	-0.81	6751.32	2.91	30.84	8.35	48.00	10.00
RTG	**93.91**	17669.59	93.45	2	-0.46	7611.35	2.32	11.73	2.54	30.00	10.00
SEA_a	**89.66**	14298.62	89.62	2	-0.04	2913.33	4.91	14.72	1.63	20.00	12.00
SEA_g	**89.24**	13979.24	89.20	2	-0.04	2902.59	4.82	14.72	1.63	20.00	12.00
AIRL	66.25	33588.54	**66.34**	1	0.09	12465.26	2.69	35.15	13.99	59.00	10.00
COVT	**92.31**	12625.36	92.14	2	-0.17	3352.87	3.77	23.10	5.04	37.00	10.00
ELEC	88.57	843.06	**88.66**	1	0.09	234.45	3.60	23.03	5.61	42.00	10.00
GMSC	**93.55**	2180.26	93.52	2	-0.03	443.70	4.91	14.10	1.61	17.00	10.00
Average	84.13	16634.58	84.06	1.71	-0.08	4986.02	3.84	22.05	5.39	36.50	10.29

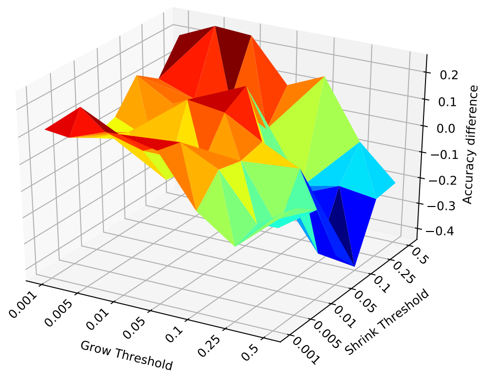
Table 6.3.4: ESRF comparison with ARF100. $T_s = T_g = 0.001$

	ARF100		ESRF accuracy			ESRF speedup		ESRF size			
Dataset	Acc(%)	Time(s)	Acc(%)	Rank	Delta	Time(s)	Speedup	mean	stdev	max	min
AGR_a	89.73	22670.86	**90.47**	1	0.74	4916.52	4.61	12.42	2.92	26.00	10.00
AGR_g	84.53	24630.29	**86.53**	1	2.00	5348.60	4.60	13.31	4.16	38.00	10.00
HYPER	85.16	20157.96	**85.38**	1	0.22	14878.47	1.35	72.03	22.60	89.00	10.00
LED_a	**73.72**	15344.18	73.67	2	-0.05	3803.60	4.03	22.97	9.03	58.00	10.00
LED_g	**72.86**	15319.00	72.74	2	-0.12	4119.97	3.72	24.75	10.60	49.00	10.00
RBF_f	**72.35**	19938.81	72.30	2	-0.05	11706.91	1.70	57.24	17.10	89.00	10.00
RBF_m	**86.01**	19638.30	85.87	2	-0.14	12117.52	1.62	61.58	19.05	89.00	10.00
RTG	**93.91**	17669.59	93.61	2	-0.30	6169.04	2.86	11.13	2.48	29.00	10.00
SEA_a	**89.66**	14298.62	89.64	2	-0.02	3665.54	3.90	21.38	6.05	38.00	10.00
SEA_g	**89.24**	13979.24	89.21	2	-0.03	3467.54	4.03	19.14	5.29	35.00	10.00
AIRL	66.25	33588.54	**66.57**	1	0.32	20700.48	1.62	66.36	20.79	89.00	10.00
COVT	92.31	12625.36	**92.34**	1	0.03	5065.27	2.49	40.24	12.57	61.00	10.00
ELEC	88.57	843.06	**88.70**	1	0.13	254.51	3.31	26.92	7.23	43.00	10.00
GMSC	**93.55**	2180.26	93.53	2	-0.02	521.10	4.18	18.30	4.44	35.00	10.00
Average	84.13	16634.58	84.33	1.57	0.19	6909.65	3.14	33.41	10.31	54.86	10.00

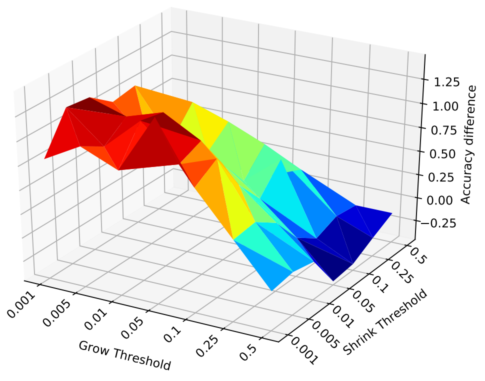
6.3.3 Elastic Random Forest

The last evaluation in this chapter assesses the performance of *ESRF* when only using the elastic component, namely the Elastic Random Forest (*ERF*). Again, results are compared to an ARF100 in terms of accuracy and ensemble size. The results for the RTG dataset are missing due to large execution times: *ERF* rapidly reaches 90 trees and further from it, many resize operations are triggered increasing the execution time considerably.

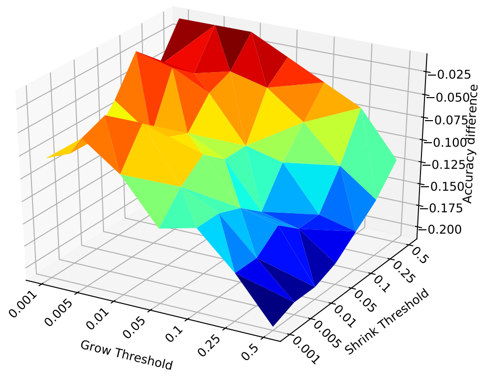
The influence of the grow and shrink thresholds in the *ERF* accuracy are



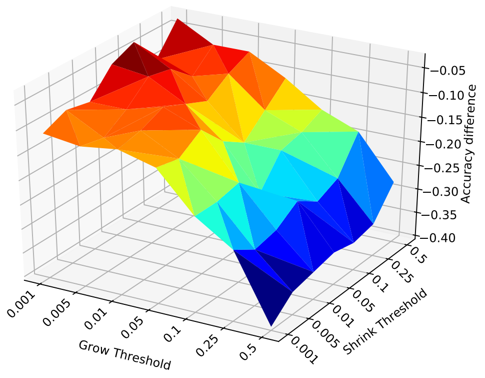
(a) AGR_a



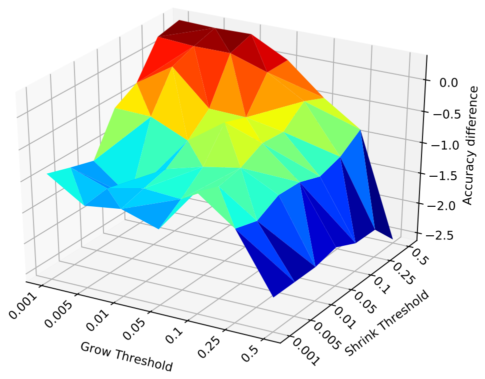
(b) AGR_g



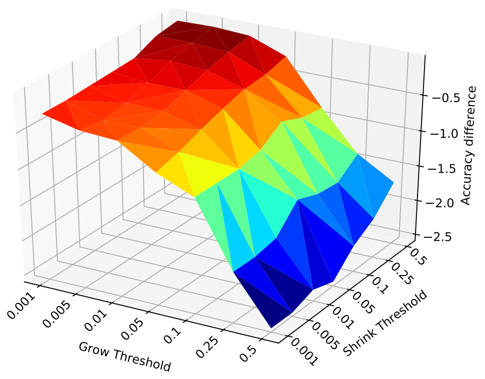
(c) LED_a



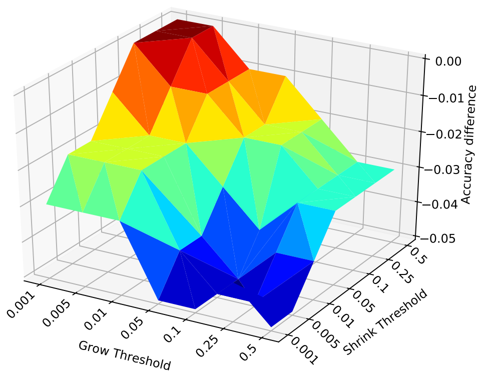
(d) LED_g



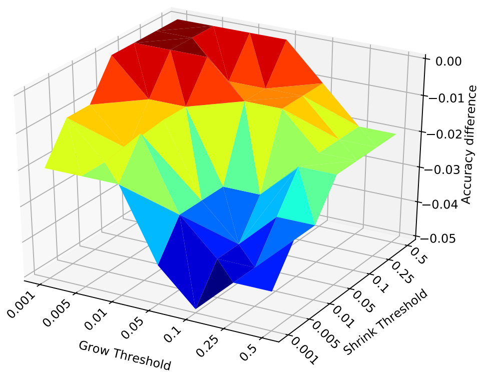
(e) RBF_f



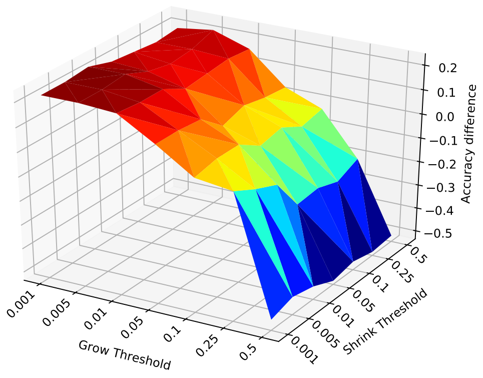
(f) RBF_m



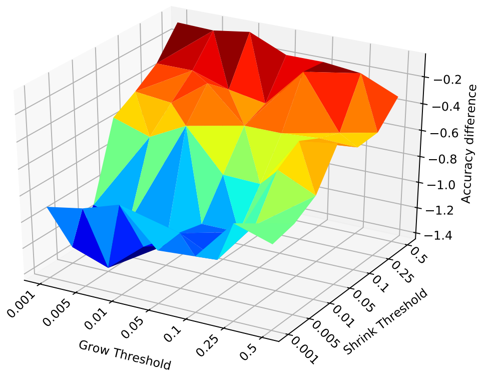
(g) SEA_a



(h) SEA_g



(i) HYPER



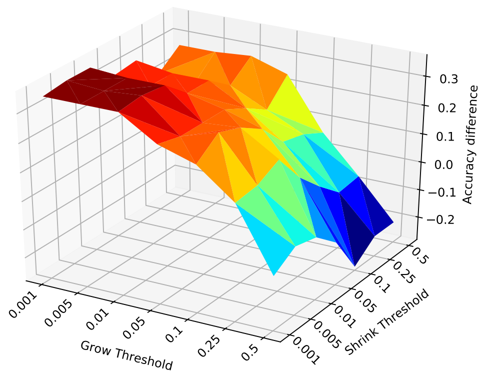
(j) RTG

Figure 6.3.4: Influence of using a resize factor $r = 5$ in the accuracy distance w.r.t. ARF100 of the grow and shrink thresholds on the synthetic datasets. Negative numbers means ARF100 is better.

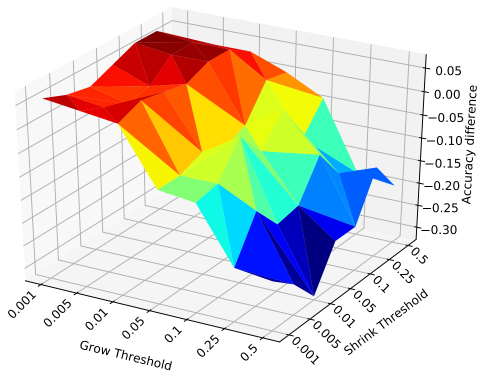
less significant than in *ESRF*: in almost all tests the *ERF* can adapt at early steps, rapidly matching the ARF100 size and accuracy.

For example, using the resource-constrained configuration ($T_g = 0.01$ and $T_s = 0.001$) the *ERF* average ensemble size is 90 with marginal differences in the accuracy with respect the *ARF100*. Observe that when using both the elastic and the swap mechanism (*ESRF*), the average ensemble size is four times lower for the same thresholds configuration (see Table 6.3.3).

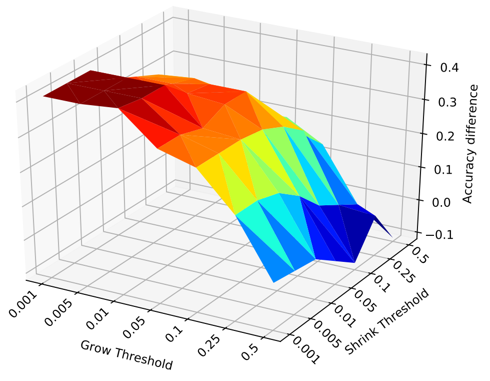
Without a component affecting the underlying ensemble behaviour (such as the swap component), the elastic component in *ERF* only switches between ARFs with different sizes. It is expected then that the *ERF* will prefer growing over shrinking since the *ARF* accuracy slowly but constantly improves when



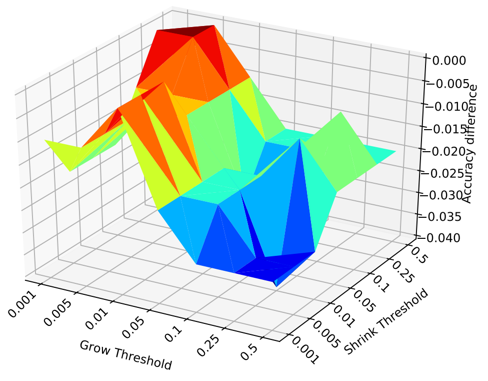
(a) AIRL



(b) COVT



(c) ELEC



(d) GMSC

Figure 6.3.5: Influence of using a resize factor $r = 5$ in the accuracy distance w.r.t. ARF100 of the grow and shrink thresholds on the real-world datasets. Negative numbers means ARF100 is better.

increasing the ensemble size from 10 to 100 (see Figure 6.1.1). Also, the *ARF* accuracy never drops significantly at any point.

Even when using a very restrictive threshold configuration such as $T_g = 0.1$ and $T_s = 0.1$, *ERF* the average ensemble size is 91. In 13 out of 14 datasets the average size is 85 or larger, indicating that the *ERF* is using high number of tree most of the time. The *ESF* accuracy difference with respect *ARF100* are marginal: -0.02 on average.

Table 6.3.5: ESRF resize factor $r = 5$ comparison with ARF100. $T_g = 0.01$ and $T_s = 0.5$

	ARF100		ESRF accuracy			ESRF speedup		ESRF size			
Dataset	Acc(%)	Time(s)	Acc(%)	Rank	Delta	Time(s)	Speedup	mean	stdev	max	min
AGR_a	89.73	22670.86	89.97	1	0.24	7020.70	3.23	25.84	9.92	50.00	10.00
AGR_g	84.53	24630.29	84.52	2	-0.01	6613.64	3.72	21.10	6.99	45.00	10.00
HYPER	85.16	20157.96	85.32	1	0.16	16212.10	1.24	79.40	18.18	89.00	10.00
LED_a	73.72	15344.18	73.71	2	-0.01	7888.49	1.95	51.48	24.24	89.00	10.00
LED_g	72.86	15319.00	72.79	2	-0.07	8691.49	1.76	56.72	22.83	89.00	10.00
RBF_f	72.35	19938.81	72.66	1	0.31	16045.70	1.24	82.10	13.09	89.00	10.00
RBF_m	86.01	19638.30	85.97	2	-0.04	15092.19	1.30	79.05	17.91	89.00	10.00
RTG	93.91	17669.59	93.85	2	-0.06	26050.77	0.68	38.96	14.00	60.00	10.00
SEA_a	89.66	14298.62	89.65	2	-0.01	4518.45	3.16	27.85	6.82	40.00	10.00
SEA_g	89.24	13979.24	89.24	1	0.00	4738.44	2.95	29.26	7.81	40.00	10.00
AIRL	66.25	33588.54	66.53	1	0.28	22724.20	1.48	74.39	19.62	89.00	10.00
COVT	92.31	12625.36	92.34	1	0.03	6768.14	1.87	56.85	22.46	89.00	10.00
ELEC	88.57	843.06	88.76	1	0.19	415.46	2.03	48.46	21.73	89.00	10.00
GMSC	93.55	2180.26	93.54	2	-0.01	736.14	2.96	28.24	9.75	50.00	10.00
Average	84.13	16634.58	84.20	1.50	0.07	10251.14	2.11	49.98	15.38	71.21	10.00

Table 6.3.6: ERF comparison with ARF100. Resource-constrained scenario: $T_g = 0.01$ and $T_s = 0.001$

	ARF100		ESRF accuracy			ESRF speedup		ESRF size			
Dataset	Acc(%)	Time(s)	Acc(%)	Rank	Delta	Time(s)	Speedup	mean	stdev	max	min
AGR_a	89.73	22670.86	89.73	1	0.00	21998.92	1.03	97.72	2.79	99.00	12.00
AGR_g	84.53	24630.29	84.81	1	0.28	24230.22	1.02	98.48	1.69	99.00	12.00
HYPER	85.16	20157.96	85.17	1	0.01	19985.14	1.01	98.96	1.37	99.00	13.00
LED_a	73.72	15344.18	73.71	2	-0.01	14913.62	1.03	98.33	4.52	99.00	12.00
LED_g	72.86	15319.00	72.84	2	-0.02	14941.38	1.03	98.46	4.50	99.00	12.00
RBF_f	72.35	19938.81	72.31	2	-0.04	19596.19	1.02	98.32	2.94	99.00	12.00
RBF_m	86.01	19638.30	86.02	1	0.01	19473.31	1.01	98.14	3.72	99	14.00
SEA_a	89.66	14298.62	89.66	1	0.00	14088.31	1.01	98.80	2.45	99.00	11.00
SEA_g	89.24	13979.24	89.25	1	0.01	13968.15	1.00	98.74	2.45	99.00	11.00
AIRL	66.25	33588.54	66.24	2	-0.01	34950.37	0.96	98.69	4.13	99.00	12.00
COVT	92.31	12625.36	92.32	1	0.01	12490.38	1.01	98.70	2.58	99.00	11.00
ELEC	88.57	843.06	88.70	1	0.13	785.47	1.07	93.65	19.99	99.00	13.00
GMSC	93.55	2180.26	93.57	1	0.02	2225.28	0.98	97.39	9.08	99.00	14.00
Average	83.38	16554.96	83.41	1.31	0.03	16434.36	1.01	90.48	4.50	91.38	11.15

6.4 Summary

This chapter presented a new ensemble method for evolving data streams: ELASTIC SWAP RANDOM FOREST (*ESRF*). *ESRF* aims at reducing the number of trees required by the reference ADAPTIVE RANDOM FOREST (*ARF*) ensemble while providing similar accuracy. *ESRF* extends *ARF* with two orthogonal components: 1) a swap component that splits learners into two sets based on their accuracy (only classifiers with the highest accuracy are used to make predictions); and 2) an elastic component for dynamically increasing or de-

Table 6.3.7: ERF comparison with ARF100. Resource-constrained scenario: $T_g = 0.1$ and $T_s = 0.1$

	ARF100		ESRF accuracy			ESRF speedup		ESRF size			
Dataset	Acc(%)	Time(s)	Acc(%)	Rank	Delta	Time(s)	Speedup	mean	stdev	max	min
AGR_a	89.73	22670.86	89.70	2	-0.03	21706.67	1.04	95.92	3.86	99.00	12.00
AGR_g	84.53	24630.29	84.43	2	-0.10	24234.53	1.02	95.92	3.86	99.00	12.00
HYPER	85.16	20157.96	85.14	2	-0.02	19898.96	1.01	98.47	1.69	99.00	13.00
LED_a	73.72	15344.18	73.69	2	-0.03	14040.89	1.09	93.01	16.46	99.00	12.00
LED_g	72.86	15319.00	72.83	2	-0.03	14109.31	1.09	93.01	16.46	99.00	12.00
RBF_f	72.35	19938.81	72.32	2	-0.03	19815.55	1.01	98.76	2.82	99.00	12.00
RBF_m	86.01	19638.30	86.00	2	-0.01	19312.92	1.02	98.64	2.34	99.00	14.00
SEA_a	89.66	14298.62	89.65	2	-0.01	11967.52	1.19	85.05	28.41	99.00	11.00
SEA_g	89.24	13979.24	89.24	1	0.00	12093.98	1.16	85.05	28.41	99.00	11.00
AIRL	66.25	33588.54	66.24	2	-0.01	32640.22	1.03	98.56	5.35	99.00	12.00
COVT	92.31	12625.36	92.32	1	0.01	12433.49	1.02	98.83	2.48	99.00	11.00
ELEC	88.57	843.06	88.58	1	0.01	731.83	1.15	87.47	28.86	99.00	13.00
GMSC	93.55	2180.26	93.55	1	0.00	1254.44	1.74	56.50	42.09	99.00	14.00
Average	83.38	16554.96	83.36	1.69	-0.02	15710.79	1.12	91.17	14.08	99.00	12.23

creasing the number of classifiers in the ensemble. The experimental evaluation of *ESRF* and comparison with the original *ARF* shows how the two new components effectively contribute to reducing the number of classifiers up to one third while providing almost the same accuracy, resulting in speed-ups in terms of per-sample execution time close to 3x. In addition, a sensitivity analysis of the two thresholds determining the elastic nature of the ensemble has been performed, establishing a trade-off in terms of resources (memory and computational requirements) and accuracy (which in all cases is comparable to the accuracy achieved by ARF100).

The next chapter presents the final contribution of this dissertation, a multithreaded ensemble for processing data streams that fully exploits modern CPUs capabilities. The proposed design can process instances in the order of few microseconds, meaning an average speedup of 85x with respect MOA.

7 Low-Latency Multi-threaded Ensemble Learning for Dynamic Big Data Streams

The last contribution in this dissertation is a high performance low-latency incremental HT and multi-threaded ARF ensemble. Modularity, scalability and adaptivity to a variety of hardware platforms, from edge to server devices, are the main requirements that have driven the proposed design.

This contribution shows the opportunities the proposed design offers in terms of optimised cache memory layout, use of vector SIMD capabilities available in functional units and use of multiple cores inside the processor. Although the parallelisation of decision trees and ensembles for batch classification has been considered in the last years, the solutions proposed do not meet the requirements of real-time streaming.

An extensive evaluation of the proposed designs, in terms of accuracy and performance, and comparison against two state–of–the–art reference implementations: MOA (Massive Online Analysis [12]) and StreamDM [40]. The proposed designs are evaluated on a variety of hardware platforms, including Intel i7 and Xeon processors and ARM–based SoC from Nvidia and Applied Micro. This chapter also shows how the proposed single decision tree behaves in low–end devices such as the Raspberry RPi3.

7.1 LMHT Design Overview

This section presents the design of *LMHT*, a Low-latency Multi-threaded Hoeffding Tree aiming at providing portability to current processor architectures, from mobile SoC to high–end multicore processors. In addition, LMHT has been designed to be fully modular so that it can be reused as a standalone tree or as a building block for other algorithms, including other types of decision trees and ensembles.

7.1.1 Tree Structure

The core of the LMHT binary tree data structure is completely agnostic with regard to the implementation of leaves and counters. It has been designed to be cache friendly, compacting in a single L1 CPU cache line an elementary binary sub-tree with a certain depth. When the processor requests a node, it fetches a cache line into L1 that contains an entire sub-tree; further accesses to the sub-tree nodes result in cache hits, minimising the accesses to main memory. For example, Figure 7.1.1 shows how a binary tree is split into 2 sub-trees, each one stored in a different cache line. In this example, each sub-tree has a maximum height of 3, thus, a maximum of 8 leaves and 7 internal nodes; leaves can point to root nodes of other sub-trees.

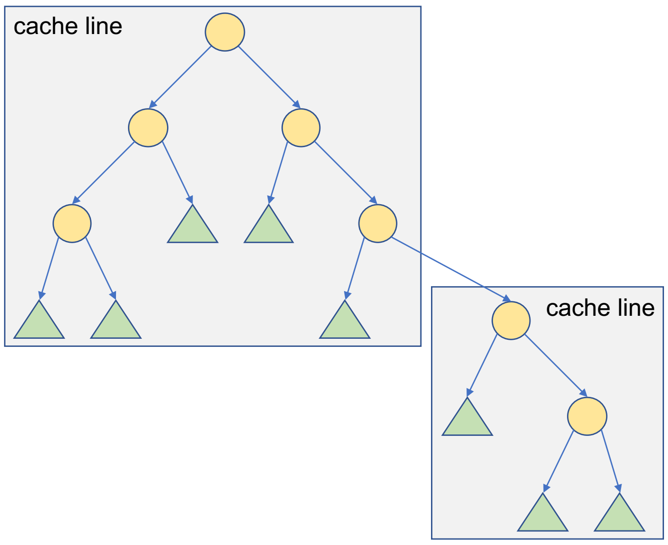


Figure 7.1.1: Splitting a binary tree into smaller binary trees that fit in cache lines

In the scope of this chapter we assume 64-bit architectures and cache line lengths of 64 bytes (the usual in Intel x86_64 and ARMv8 architectures today). Although 64 bits are available only 48 bits are used to address memory, leaving 16 bits for arbitrary data. Based on that we propose the cache line layout shown in Figure 7.1.2: 8 consecutive rows, each 64 bits wide storing a leaf flag (1 bit), an attribute index (15 bits) and a leaf pointer address (48 bits).

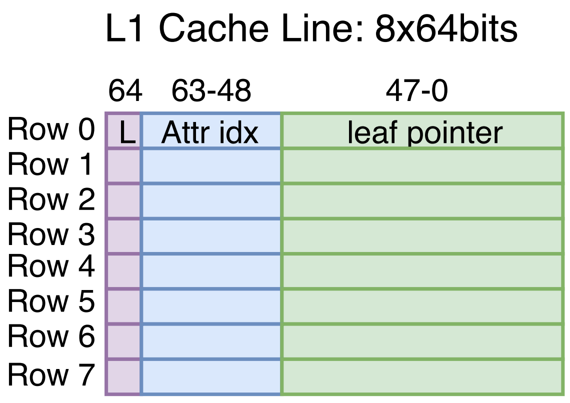


Figure 7.1.2: Sub-tree L1 cache line layout

With this layout a cache line can host a sub-tree with a maximum height of 3 (8 leaves and 7 internal nodes, as the example shown in Figure 7.1.2). The 1-bit leaf flag informs if the 48-bit leaf pointer points to the actual leaf node data structure (where all the information related with the leaf is stored) or points to the root node of the next sub-tree. The 15-bit attribute index field indexes the attribute that is used in each one of the 7 possible internal nodes. This imposes a maximum of 2^{15} (32,768) combinations (i.e. attributes per

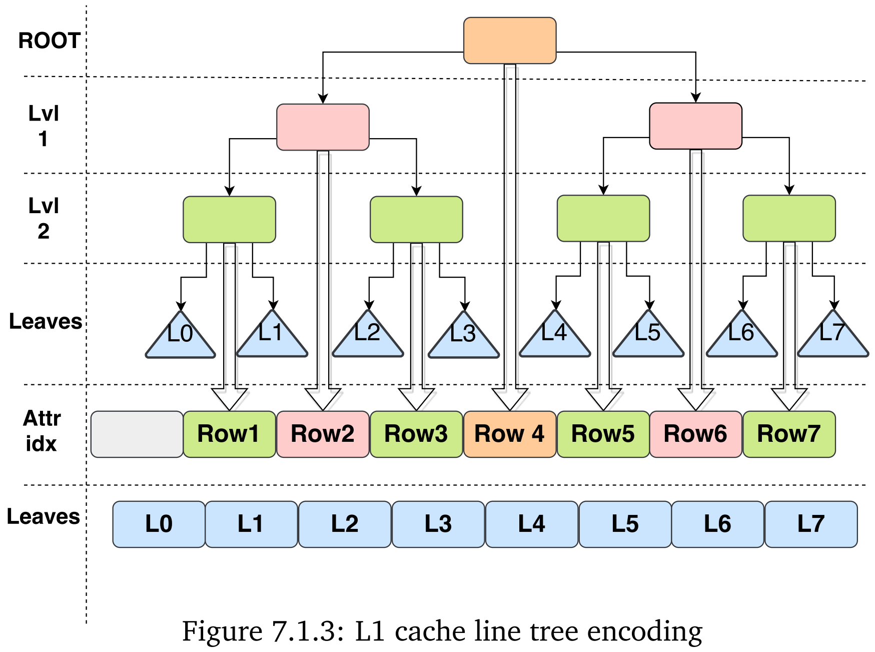


Figure 7.1.3: L1 cache line tree encoding

instance), one of them reserved to indicate that a sub-tree internal node is the last node in the tree traversal. For current problem sizes we do not expect this number of attributes to be a limiting factor. Having an invalid attribute index allows sub-trees to be allocated entirely and internally grow in an incremental way as needed.

The specific mapping (encoding) of the sub-tree into this 64-byte cache line layout is shown in Figure 7.1.3. Regarding attributes, the root node attribute index is stored in row 4, level 1 attributes are stored in rows 2 and 6, and level 2 attributes are stored in rows 1, 3, 5 and 7; the attribute index in row 0 is left unused. Regarding leaf pointers, they are mapped (and accessed) using a 3-bit lookup value in which each bit represents the path taken at each sub-tree level: the most significant bit is root node, next bit is the attribute in level 1, and the least significant bit represents the attribute at level 2. The bit is true if at that level the traverse took the left child, and false otherwise. The resulting value is used as the row index (offset) to access to the leaf pointer column.

7.1.2 Leaves and Counters

Each leaf node in the HT points to an instance of the data structure that encapsulates all the information that is required to compute its own split criterion function (G in Algorithm 1) and apply a leaf classifier; the design for these two functionalities is based on templates and polymorphism in order to provide the required portability and modularity. The key component in the proposed design are the leaf counters, which have been arranged to take benefit of the SIMD capabilities of nowadays core architectures.

For each label j ($0 \leq j < L$) one needs to count how many times each attribute i in the leaf ($0 \leq i < N$) occurred with each one of its possible values k ($0 \leq k < V_i$). This requires $L \times \sum_{i=0}^{N-1} V_i$ counters in total. For simplicity, in this chapter we use binary attribute counters (though there is no reason why other attribute counters could not be implemented) and no missing attributes in the input instances. Therefore, for each label j one only needs to count how many times attribute i had value 1 and the total number of attributes seen for that label (in order to determine how many times each attribute i had value 0). With these simplifications $L \times (N + 1)$ counters are needed in total.

Attribute counters are stored consecutively in memory for each label, each one occupying a certain number of bits (32 bits in the implementation in this chapter). This layout in memory allows the use of SIMD registers and instructions available in current processors. For example Intel AVX2 [63] can accommodate 8 32-bit counters in each SIMD register and operate (sum for example) them in parallel. The proposed layout allows the use of both vertical (between two SIMD registers, e.g. the same attribute for different labels) and horizontal (inside one SIMD register, e.g. different attributes or values for the same attribute for the same label) SIMD instructions. These are the operations needed to perform the additions, multiplications and divisions in expression 2.3 (the logarithm that is needed to compute the entropy is not available in current SIMD instruction sets). The computation of the related Naive Bayes classifier is also very similar in terms of operations required, so it also benefits from SIMD in the same way.

7.2 Multithreaded Ensemble Learning

This section presents the design of a multithreaded ensemble for data streams. Our proposed design is very similar to the *Adaptive Random Forest* (ARF) used in Chapter 6, however, at the time of starting this project, ARF was not published. The main differences between our proposed design and the ARF are in the strategies for dealing with concept drifting and combining votings. When a concept drift is detected, our design resets a tree while ARF uses two thresholds for dealing with concept drifting: a permissive threshold warns a drift and starts a local background learner which ends-up substituting the original learner when the drift is confirmed. Also, in our design votings are combined using the same weight for all learners, while in ARF a learner is weighted using its accuracy.

The ensemble is composed of L learners, each one making use of the *randomHT* described in the previous section. The overall design aims to low-latency response time and good scalability on current multi-core processors, also used in commodity low-end hardware.

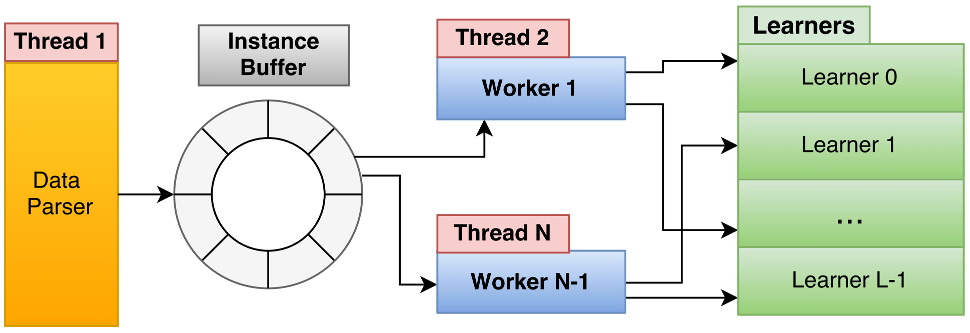


Figure 7.2.1: Multithreaded ensemble design

The proposed multithreaded implementation makes use of N threads, as shown in Figure 7.2.1: thread 1, the *Data Parser* thread, is in charge of parsing the attributes for each input sample and enqueuing into the *Instance Buffer*; threads 2 to N, the so-called *Worker* threads, execute the learners in parallel to process each of the instances in the *Instance Buffer*. The number of threads N is either the number of cores available in the processor, or the number of hardware threads the processor supports in case hyper-threading is available and enabled.

7.2.1 Instance Buffer

The key component in the design of the multithreaded ensamble is the *Instance Buffer*. Its design has been based on a simplified version of the LMAX disruptor [105], a highly scalable low-latency *ring buffer* designed to share data among threads.

In LMAX each thread has a *sequence number* that it uses to access the ring buffer. LMAX is based on the *single writer principle* to avoid writing contention: each thread only writes to its own sequence number, which can be read by other threads. Sequence numbers are accessed using atomic operations to ensure atomicity in the access to them, enabling the *at least one makes progress* semantics typically present on lock-less data structures.

Figure 7.2.2 shows the implementation of the Instance Buffer as an LMAX *Ring Buffer*. The *Head* points to the last element inserted in the ring and it is only written by the data parser thread, adding a new element in the ring if and only if $Head - Tail < \#slots$. Each worker thread i owns its $LastProcessed_i$ sequence number, indicating the last instance processed by worker i. The parser thread determines the overall buffer *Tail* using the circular lowest $LastProcessed_i$ for all workers i.

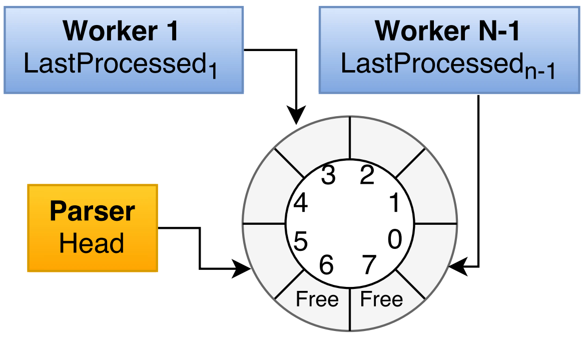


Figure 7.2.2: Instance buffer design

Atomic operations have an overhead: require fences to publish a value written (order non-atomic memory accesses). In order to minimise the overhead introduced, our proposed design allows workers to obtain instances from the *Ring Buffer* in batches. The batch size is variable, depending on the values of each worker $LastProcessed_i$ and *Head*.

7.2.2 Random Forest Workers and Learners

Random Forest learners are in charge of sampling the instances in the *Instance Buffer* with repetition, doing the *randomHT* inference and, if required, reseting a learner when drift is detected. Each worker thread has a number of learners ($\frac{|L|}{|N-1|}$ approximately) assigned in a static way (all learners l such that $l\%(N-1) = i$, being i the worker thread identifier). This static task distribution may introduce certain load unbalance but avoids the synchronisation that would be required by a dynamic assignment of learners to threads. In practice, we do no expect this unbalance to be a big problem due to the randomisation present in both the sampling and in the construction of the *randomHT*.

Each entry in the *Ring Buffer* stores the input instance and a buffer where each worker stores the output of the classifiers. To minimise the accesses to this buffer, each worker locally combines the output of its assigned learners for each instance; once all learners assigned to the worker are finished, the worker writes the combined result into the aforementioned buffer. Finally the data parser thread is responsible of combining the outputs produced by the workers and generating the final output.

7.3 Implementation Notes

This section includes some implementation notes that may be useful for someone willing to use the design proposed in this chapter or extend its functionalities to implement other learners based on classification trees.

All the implementation relies on C++14 powerful template features. Templates replace dynamic allocations at runtime by static objects that can be optimized by the compiler. Attribute counters are designed using a relaxed version of the *Single Responsibility Principle* (SRP [82]). The counter class only provides the data, and any extra functionalities such as the split criterion (Information Gain in this study) or leaf classifier are implemented in a separate object. Information gain and leaf classifiers rely on the compiler for the automatic vectorisation of the computation in the HT leaves as a fast way to achieve SIMD portability.

For safe access to the instance buffer in the multithreaded implementation of Random Forest, the implementation makes use of the C++11 atomic API (`std::memory_order`), allowing to fine tune the order of memory accesses in a portable way. In particular, the use of the *memory_order_consume* for write operations and *memory_order_relaxed* for read operations. Regarding threads, although the C++11 `std::thread` offers a portable API across platforms, pinning threads to cores must be done using the native thread library (e.g. Pthreads on Linux); thread pinning has been required to improve scalability in some platforms.

7.4 Experimental Evaluation

This section evaluates the proposed design and implementation for both LMHT and the multithreaded RF ensemble.

Performance and accuracy are compared against two state–of–the–art reference implementations: MOA (Massive Online Analysis [12]) and StreamDM

[40]. StreamDM does not provide a RF implementation, but we considered it in the single HT evaluation since it is also written in C++.

In order to setup a basic environment for the evaluations, we implemented binary counters for the HT which can only process binary attributes. This way, all datasets used in the evaluations were binarized using the WEKA [33] unsupervised *NumericToBinary* filter with the default values. The resulting binarized datasets have $90+$ binary attributes (except those generated with LED and SEA dataset which have 25 binary features each). Table 7.4.1 summarises the datasets used in this section and their resulting characteristics.

Table 7.4.1: Datasets used in the experimental evaluation, including both real world and synthetic datasets

Dataset	Samples	Attributes	Labels	Generator
RBF1-6	1,000,000	91	5	RandomRBF Drift
HYPER1-2	1,000,000	91	5	Hyperplane
LED1	1,000,000	25	10	LED Drift
SEA1-2	1,000,000	25	2	SEA
COVT	581,012	134	7	Real world
ELEC	45,312	103	2	Real world

The hardware platforms that have been used to conduct the accuracy and performance analysis in this section are summarised in Table 7.4.2. A desktop-class Intel I7 platform is used to compare accuracy and to evaluate the performance of the two reference platforms, StreamDM and MOA (running on Oracle JAVA JDK 1.8.0_73). In order to perform a more complete evaluation of the throughput and scalability achieved by LMHT, additional platforms have been used, including three ARM-based systems, from low-end Raspberry Pi3 to NVIDIA Jetson TX1 embedded system and Applied Micro X-Gene 2 server board. Finally, a system based on the latest Intel Xeon generation sockets has been used to explore the scalability limits of the multithreaded RF.

7.4.1 Hoeffding Tree Accuracy

Table 7.4.3 compares the accuracy achieved by MOA, StreamDM and LMHT. The main conclusion is that the three implementations behave similarly, with less than one percent difference in accuracy in all datasets but RBF2 and RBF5 for which LMHT improves almost two percent. On the real world data sets (COVT and ELEC) learning curves are almost identically for all three implementations (not included for page limit reasons).

The minor differences that are observed are due to the fact that in few cases LMHT obtains different values for the Hoeffding Bound (eq. 2.4) at the same time step when compared to MOA, and this may cause node splits at different time steps (and in few cases using different attributes). MOA uses

Table 7.4.2: Platforms used in the experimental evaluation

Platform	Intel Xeon	Intel i7	X-Gene2	Jetson TX1	Raspberry RPi3
Processor	Xeon Platinum 8160	i7-5930K	ARMv8-A	Cortex A57	Cortex A53
Clock Speed	2.1Ghz	3.7 Ghz	2.4Ghz	1.9Ghz	1.2Ghz
Cores	24	6	8	4	4
RAM	96GB	64GB	128GB	4GB	1GB
Storage	Network (gpfs)	SSD	Network (gpfs)	eMMc	Class 10 SD Card
SO	SUSE 12 server	Debian 8.7	Debian 8	L4T	Fedora 26 64Bits
Kernel	4.4.59	4.7.8-1	4.3.0	3.10.96	4.11.8-300
Compiler	GCC 7.1.0	GCC 6.3.0	GCC 6.1.0	GCC 6.1.0	GCC 7.1.0

Table 7.4.3: Single Hoeffding Tree accuracy comparison

Dataset	MOA	StreamDM	LMHT
HYPER1	**84.67**	**84.67**	**84.67**
HYPER2	**78.03**	**78.03**	**78.03**
LED1	**68.58**	**68.58**	68.40
RBF1	82.04	82.04	**82.98**
RBF2	43.71	43.71	**45.86**
RBF3	31.58	31.58	**32.24**
RBF4	82.04	82.04	**82.98**
RBF5	75.88	75.88	**77.40**
RBF6	73.71	73.71	**74.61**
SEA1	85.81	85.81	**85.85**
SEA2	85.75	85.75	**85.76**
COVT	**73.18**	**73.18**	73.16
ELEC	**79.14**	**79.14**	**79.14**

dynamic vectors to store the attribute counters. These counters are forwarded to child nodes as the previous class distribution in the presence of a split. LMHT uses a preallocated vector that can only grow. In some cases these vectors can have different sizes at the same time step, affecting the class range used to compute the bound.

7.4.2 Hoeffding Tree Throughput Evaluation

Table 7.4.4 shows the throughput (instances per millisecond) achieved by the two reference implementations and LMHT on the different platforms, for each dataset and the average of all datasets. For each implementation/platform, the speedup with respect to MOA is also shown.

On the Intel i7-based platform LMHT outperforms the two reference implementations by a 6.7x factor, achieving on the average a throughput above the 500 instances per millisecond. The worst case (COVT) has a throughput close to 250 instances (of 134 attributes) per millisecond. StreamDM per-

Table 7.4.4: Single Hoeffding Tree throughput (instances per ms) on Intel (top) and ARM (bottom) compared to MOA. ↓ indicates speed-down (MOA is faster)

	Intel				
	MOA(i7)	StreamDM (i7)		LMHT(i7)	
Dataset	inst/ms	inst/ms	Speedup	inst/ms	Speedup
HYPER1	62.11	54.79	0.88 ↓	418.94	6.74
HYPER2	61.14	65.04	1.06	416.67	6.81
LED1	141.20	99.18	0.70 ↓	834.03	5.91
RBF1	51.56	41.96	0.81 ↓	333.00	6.46
RBF2	52.23	42.76	0.82 ↓	333.56	6.39
RBF3	56.06	42.98	0.77 ↓	332.78	5.94
RBF4	54.54	42.79	0.78 ↓	334.56	6.13
RBF5	54.52	42.17	0.77 ↓	326.90	6.00
RBF6	53.69	41.83	0.78 ↓	332.56	6.19
SEA1	192.68	162.81	0.84 ↓	1253.13	6.50
SEA2	179.22	166.09	0.93 ↓	1250.00	6.97
COVT	41.20	34.65	0.84 ↓	251.63	6.11
ELEC	36.70	60.98	1.66	415.71	11.33
Average	79.76	69.08	0.90 ↓	525.65	6.73

	Intel	**ARM**					
	MOA(i7)	LMHT(Jetson)		LMHT(X-Gene2)		LMHT(RPi3)	
Dataset	inst/ms	inst/ms	Speedup	inst/ms	Speedup	inst/ms	Speedup
HYPER1	62.11	127.96	2.06	101.18	1.63	89.57	1.44
HYPER2	61.14	128.49	2.10	127.16	2.08	89.39	1.46
LED1	141.20	277.01	1.96	231.27	1.64	79.81	0.57 ↓
RBF1	51.56	104.98	2.04	52.74	1.02	43.43	0.84 ↓
RBF2	52.23	107.54	2.06	94.71	1.81	42.84	0.82 ↓
RBF3	56.06	110.06	1.96	94.55	1.69	42.60	0.76 ↓
RBF4	54.54	110.84	2.03	95.30	1.75	43.43	0.80 ↓
RBF5	54.52	110.56	2.03	95.05	1.74	43.24	0.79 ↓
RBF6	53.69	110.24	2.05	94.81	1.77	42.95	0.80 ↓
SEA1	192.68	401.61	2.08	398.57	2.07	281.53	1.46
SEA2	179.22	402.58	2.25	398.57	2.22	281.69	1.57
COVT	41.20	78.67	1.91	68.13	1.65	43.78	1.06
ELEC	36.70	122.80	3.35	87.47	2.38	86.80	2.37
Average	79.76	168.72	2.14	149.19	1.80	93.16	1.13

forms the worst in almost all datasets, with an average slowdown compared to MOA of 0.9.

On the Jetson and X-Gene2 ARM-based platforms, LMHT performs quite similarly, achieving 3x lower throughput, on the average, compared to the Intel i7-based system. However, on the average LMHT is able to process 168 and 149 instances per millisecond on these two ARM platforms, which is better than the two reference implementations on the Intel i7, and in particular 2x better than MOA. The last column in Table 7.4.4 corresponds to the RPi3 throughput, which is similar to MOA on the Intel i7, showing how the imple-

Table 7.4.5: Comparing LMHT parser overhead (instances per ms). *Parser* includes time to parse and process input data; *No Parser* means data is already parsed in memory.

Dataset	Parser	No Parser	speedup
HYPER1	418.94	1647.45	3.93
HYPER2	416.67	1636.66	3.93
LED1	834.03	1550.39	1.86
RBF1	333.00	890.47	2.67
RBF2	333.56	878.73	2.63
RBF3	332.78	881.83	2.65
RBF4	334.56	889.68	2.66
RBF5	326.90	884.96	2.71
RBF6	332.56	875.66	2.63
SEA1	1253.13	4000.00	3.19
SEA2	1250.00	4000.00	3.20
COVT	251.63	859.49	3.42
ELEC	415.71	1618.29	3.89
Average	525.65	1585.66	3.03

mentation of LMHT is portable to low-end devices doing real-time classification on the edge.

The results are summarized in terms of performances in Figure 7.4.1, and in terms of throughput in Figure 7.4.2. Up to this point, the main factor limiting the performance of LMHT on a single HT is the CSV parser, which is in charge of reading from a file the attributes for each input sample. In order to disect the influence of this parser, Table 7.4.5 shows the overhead introduced by the parser when data is read from a file or when data is already parsed and directly streamed from memory, resulting in an average 3x improvement.

7.4.3 Random Forest Accuracy and Throughput

In this section we compare the performance and accuracy of MOA and the proposed RF ensemble design with 100 learners. Table 7.4.6 compares the accuracy of the two implementations, with less than one percentage points of difference in the average accuracy. The comparison with StreamDM is not possible since it does not provide an implementation for RF. The same table also shows the numerical stability of LMHT, with a small standard deviation (on 12 runs). These variations in LMHT are due to the random number generator used at the sampling and random attributes selection. LMHT uses a different seed at each execution, while MOA uses a default seed (unless a custom one is specified by the user; we used the default seed in MOA).

As with the single HT, learning curves for the real world datasets COVT

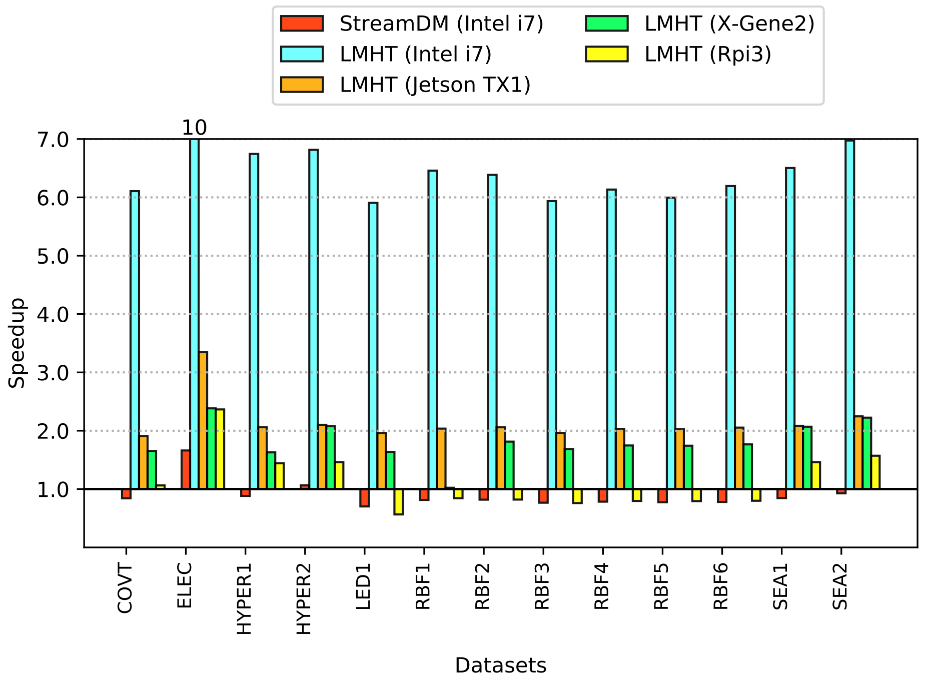


Figure 7.4.1: LMHT and StreamDM speedup over MOA using a single HT (Intel i7). Down bars mean speed-down (slower than MOA)

Table 7.4.6: Random Forest Accuraccy

	MOA	LMHT	
Dataset		Avg.	Std. dev.
HYPER1	87.97	**88.41**	0.24
HYPER2	**83.18**	82.48	0.24
LED1	**68.68**	68.18	0.20
RBF1	86.35	**87.39**	0.13
RBF2	65.96	**68.04**	0.14
RBF3	40.61	**45.05**	0.12
RBF4	86.35	**87.41**	0.13
RBF5	81.42	**82.79**	0.12
RBF6	79.10	**79.81**	0.12
SEA1	86.49	**86.54**	0.24
SEA2	86.48	**86.53**	0.24
COVT	85.34	**86.37**	0.23
ELEC	**82.41**	82.12	0.22

and ELEC have a similar pattern, as shown in Figure 7.4.3: at early stages LMHT is slightly better, but soon they become very similar.

Table 7.4.7 summarises the results in terms of throughput, comparing

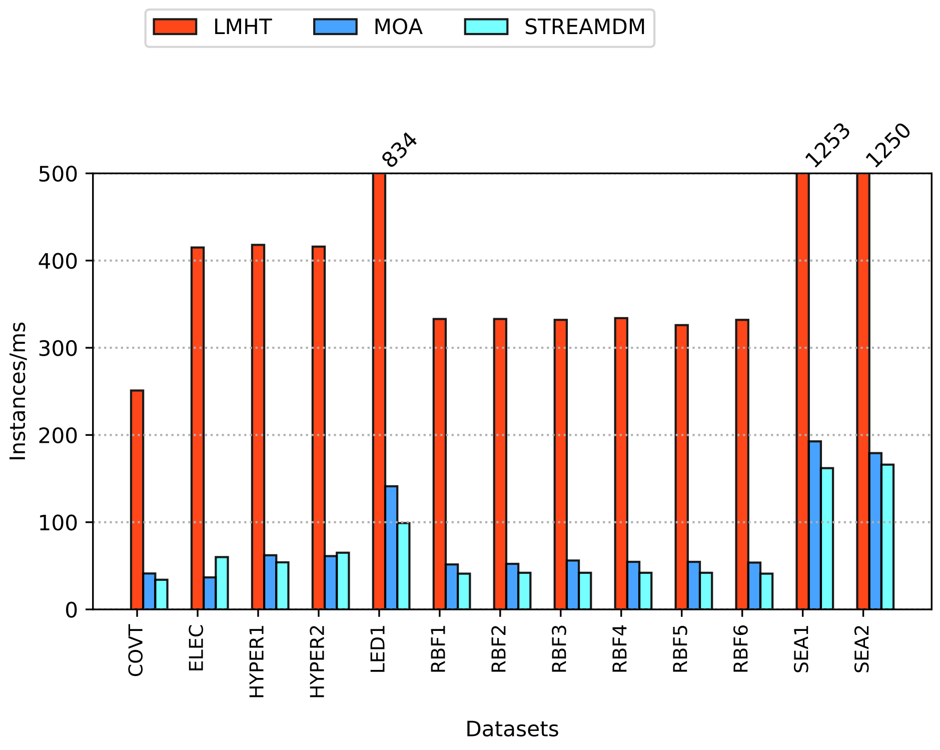
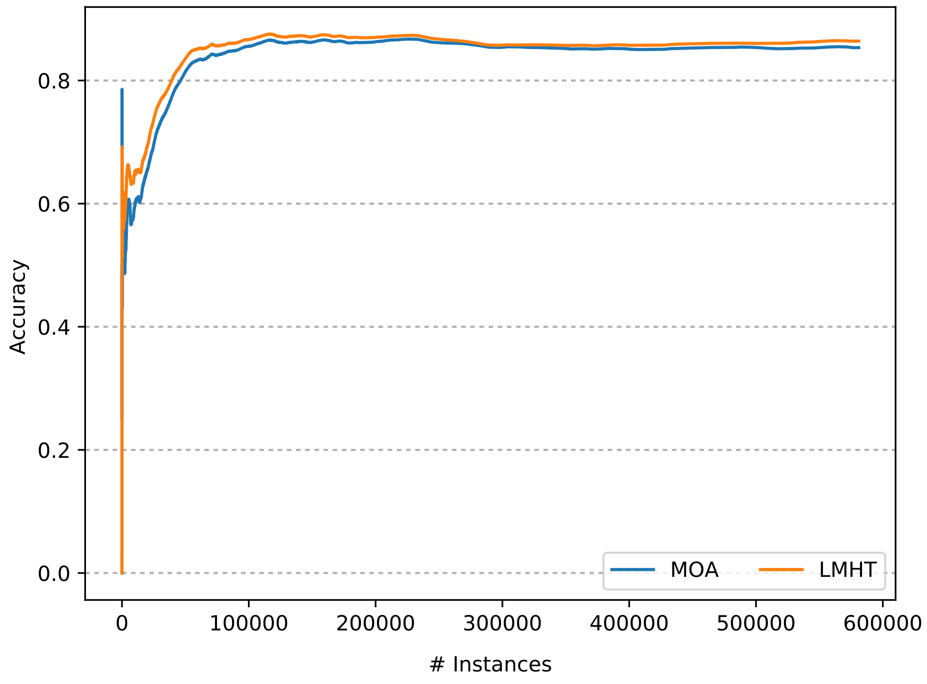


Figure 7.4.2: LMHT, StreamDM and MOA single HT throughput comparison (instances/ms)

again with the performance that the MOA reference implementation provides. On the same hardware platform (Intel i7) we observe an average throughput improvement of 85x compared to MOA when 11 threads are used as workers, resulting on an average throughput very close to 100 instances per millisecond; MOA throughput is less than 2 instances per millisecond in all tests. The Intel Xeon platform results in almost the same throughput than the Intel i7 which uses a much modest core count (6 instead of 24). Two main reasons for this behaviour: 1) the parser thread reads data from a CSV file stored in GPFS on a large cluster with several thousand nodes; if the parser thread directly streams data from memory, the throughput that is obtained raises to 175 instances per millisecond (143x faster than MOA). And 2) the different clock frequencies at which the i7 and Xeon sockets operate (3.7 and 2.1 GHz, respectively, as shown in Table 7.4.2); in any case, the Xeon–based platform allows us to do an scalability analysis up to a larger number of cores.

On the ARM-based platforms we observe improvements of 10x and 20x on the Jetson TX1 and X-Gene2 platforms, respectively.



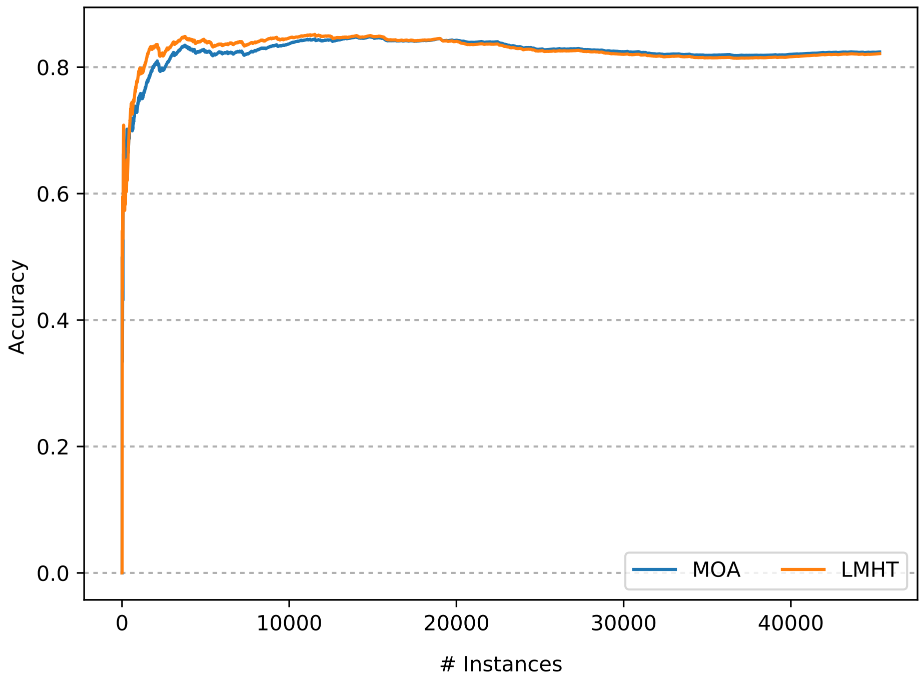


Figure 7.4.3: Random Forest learning curve: COVT (top) and ELEC (bottom)

Table 7.4.7: Random Forest throughput comparison (instances/ms)

	Intel			ARM					
	MOA(i7) 1 Worker	LMHT(i7) 11 Workers		LMHT(Jetson) 23 Workers		LMHT(X-Gene2) 3 Workers		LMHT(RPi3) 7 Workers	
Dataset	inst/ms	inst/ms	Speedup	inst/ms	Speedup	inst/ms	Speedup	inst/ms	Speedup
HYPER1	1.15	103.42	90.25	100.63	87.82	12.51	10.92	28.52	24.89
HYPER2	1.37	98.89	72.23	97.99	71.57	11.19	8.18	26.95	19.68
LED1	1.54	100.95	65.56	142.11	92.29	12.07	7.84	26.99	17.53
RBF1	0.93	103.00	111.29	105.99	114.53	13.17	14.23	29.49	31.87
RBF2	1.15	101.54	87.97	104.55	90.57	13.05	11.30	29.21	25.31
RBF3	1.72	99.86	57.91	103.36	59.93	12.70	7.36	28.49	16.52
RBF4	0.91	102.70	113.07	103.64	114.10	13.17	14.50	29.76	32.77
RBF5	0.92	102.09	111.47	104.03	113.58	13.13	14.33	29.32	32.02
RBF6	0.94	103.08	109.72	106.76	113.63	13.14	13.98	29.49	31.39
SEA1	1.71	127.39	74.29	131.08	76.44	12.36	7.21	30.76	17.94
SEA2	1.74	124.64	71.67	127.00	73.03	11.54	6.64	30.20	17.37
COVT	1.30	96.48	74.22	90.85	69.89	14.68	11.29	31.31	24.09
ELEC	1.48	109.71	74.03	97.87	66.04	15.04	10.15	23.32	15.74
Average	1.30	105.67	85.67	108.91	87.95	12.90	10.61	28.76	23.62

7.4.4 Random Forest Scalability

Finally, this subsection analyses the scalability of the proposed ensemble implementation with the number of worker threads, always limiting the analysis to the number of cores (hardware threads) available in a socket. On the commodity Intel i7 platform, LMHT achieves a relative speedup with respect to single threaded execution, between 5-7x when using 11 workers (12 threads), as shown in Figure 7.4.4. It is interesting to observe the drop in performance observed when going from 5 to 6 worker threads. Since the i7-5930K processor has 6 cores and 12 threads (two threads mapped on the same physical core), when 6 workers are used they start competing for the same physical cores introducing some work imbalance. However, the hyperthreading capabilities of the i7-5930K are able to mitigate this as the number of threads tends to the maximum hardware threads.

X-Gene2 scalability has some variability for the different datasets and speed-ups in the range 5-6.5x when using 7 worker threads (Figure 7.4.6). On the other side, Jetson achieves an almost linear speedup very close to 3x when using 3 threads as workers (Figure 7.4.5).

In order to better study the scalability limits of the LMHT ensemble, we have extended our evaluation to one of the latest Intel Xeon Scalable Processor, the Platinum 8160 socket which include 24 cores. To better analyse if the limits are in the producer parser thread or in the implementation of the instance buffer and worker threads, we consider two scenarios: parser thread reading instances from storage and directly streaming them from memory.

The two plots in Figure 7.4.7 show the speed-up, with respect to a single worker, that is achieved when using up to 24 cores (23 worker threads). With

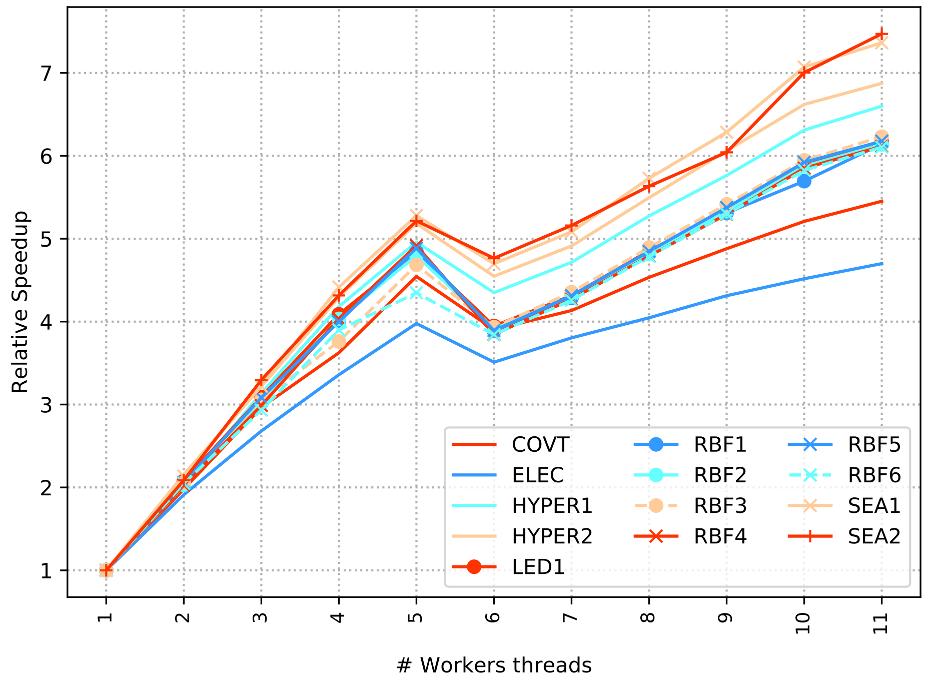


Figure 7.4.4: Random Forest relative speedup on Intel i7

the parser from storage (top plot), an speed-up between 6-11x is obtained when using 23 worker threads, which corresponds to a parallel efficiency below 45%. Both figures raise to 10-16x and 70% when the parser streams directly from memory (bottom plot).

Scalability curves in all tests have the similar shape, and the only different is the maximum scalability achieved. This may be due the nature of the different datasets affecting how our architecture fully exploits all cores. However, validating this hypotheses requires an in deep study which is out of the scope of this thesis and it is delayed to future work.

7.5 Summary

This chapter presented a novel design for real-time data stream classification, based on a Random Forest ensemble of randomised Hoeffding Trees. This work goes one big step further in fulfilling the low-latency requirements of today and future real-time analytics. Modularity and adaptivity to a variety of hardware platforms, from server to edge computing, has also been considered as a requirement driving the proposed design. The design favours an effective use of cache, SIMD units and multicores in nowadays processor sockets.

Accuracy of the proposed design has been validated with two reference implementations: MOA (for HT and Random Forest) and StreamDM (for HT).

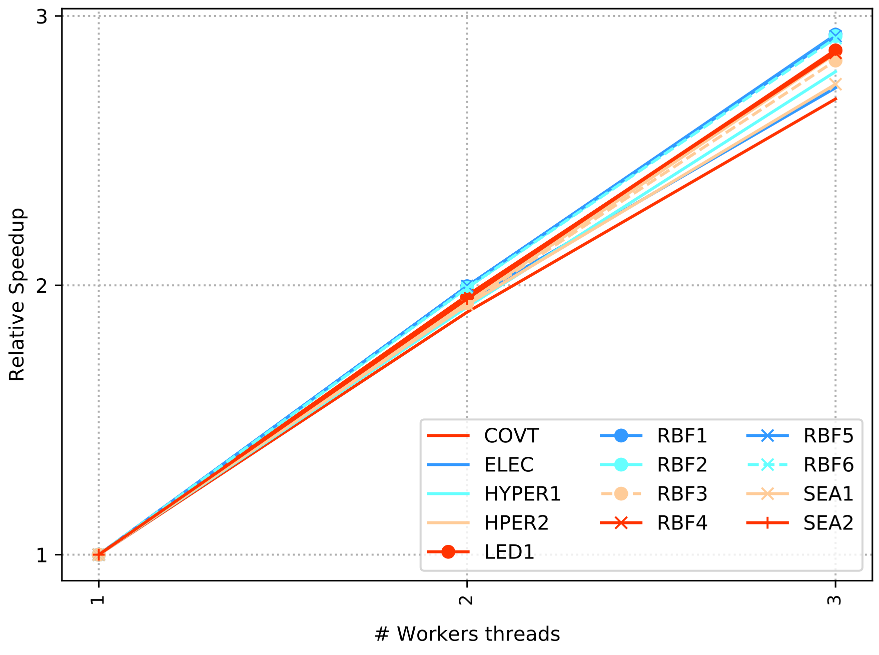


Figure 7.4.5: Random Forest relative speedup on Jetson TX1

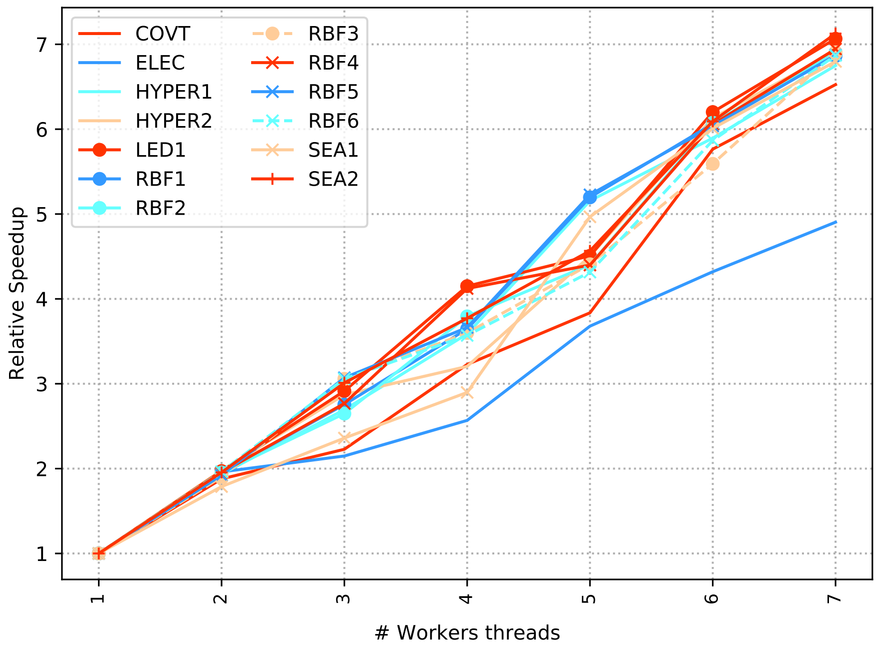
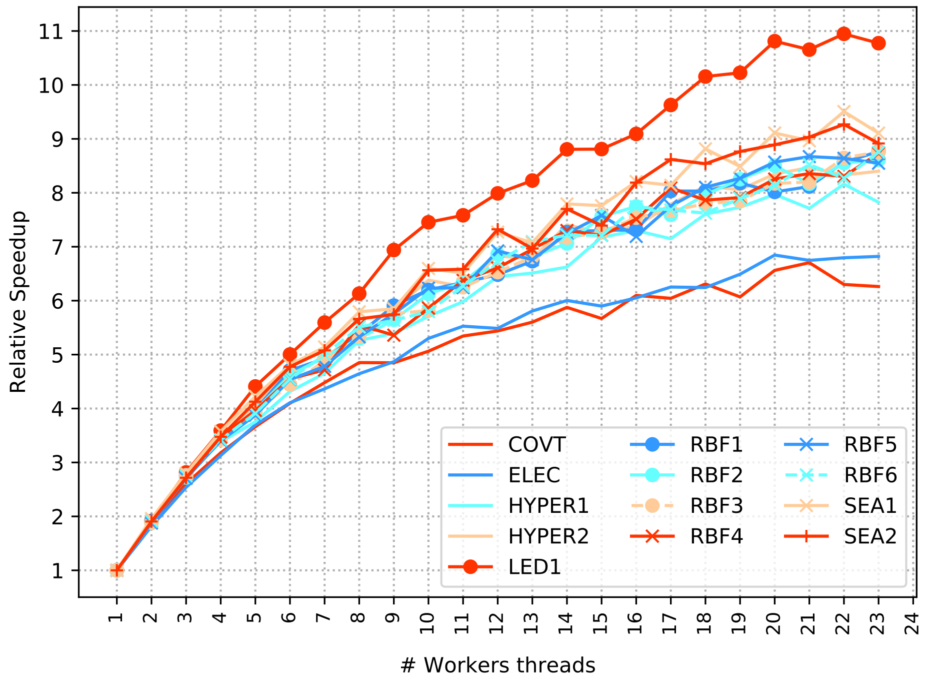


Figure 7.4.6: Random Forest speedup on X-Gene2



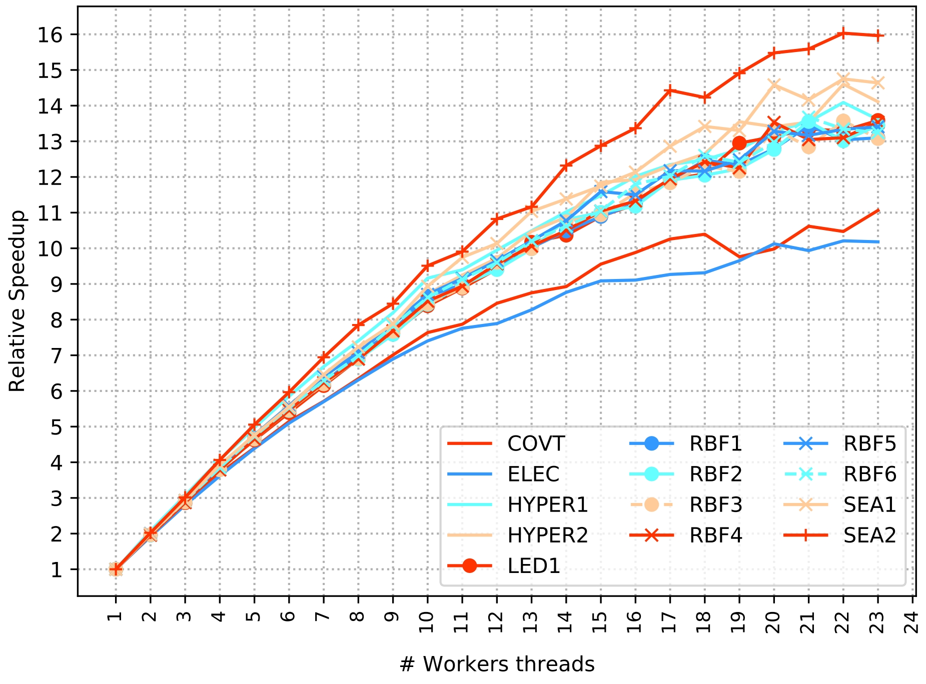


Figure 7.4.7: Intel Xeon Platinum 8160 scalability, with the parser thread streaming from storage (top) and memory (bottom).

Throughput is evaluated on a variety of platforms. On Intel-based systems: i7 desktop (6 cores) and Xeon server (24 cores) class sockets. And on ARM-based systems: NVIDIA Jetson TX1 (4 cores), Applied Micro X-Gene2 (8 cores) and low-end Raspberry RPi3 (4 cores). For single HT the performance evaluation in terms of throughput reports 6.7x (i7), around 2x (Jetson TX1 and X-Gene2) and above 1x (RPi3) compared to MOA executed in i7. For Random Forest the evaluation reports throughput improvements of 85x (i7), 87x (Xeon parsing from memory), 10x (Jetson TX1) and 23x (X-Gene2) compared to single-threaded MOA on i7. The proposed multi-threaded implementation for the ensemble shows good scalability up to the largest core count socket that we have evaluated (75% parallel efficiency when using 24 cores on the Intel Xeon).

The evaluation also reports how the parser thread, in charge of feeding instances to the HT and ensemble, can easily limit throughput. The limits are mainly observed because of the media used to store the data (GPFS, solid–state disks, eMMC, SD, ...) that feeds the learners. For large core counts, we need to investigate if the proposed single–parser design limits scalability and find the appropriate number of learners per parser ratio.

8 Conclusions

This dissertation focus on improving the performance of the Hoeffding Tree (HT) and its ensembles combinations by 1) improving single tree accuracy (Echo State Hoeffding Tree), 2) reducing ensembles of HTs resource consumption (Resource-Aware Elastic Swap Random Forest), and 3) providing a high performant and scalable ensemble of HTs design that is able to scale linearly and process instances in the order of microseconds on a desktop class intel CPU (Ultra-low latency Random Forest).

This final chapter first reviews all the important results achieved in this dissertation, and later discusses future work that will follow from our research.

8.1 Summary of Results

Data Stream Classification using Random Features

We proposed a random layer for Neural Networks (NNs) that can turn a simple gradient descent learner into a competitive method for continual learning of data streams. We highlighted important issues of using NNs for processing data streams, being the most critical one the fact that NNs are very sensitive to hyper-parameters configuration, requiring many combinations to be tested in order to achieve competitive accuracy; this is not feasible in a data streams setup, and solving this issue is out of the scope of this dissertation.

Echo State Hoeffding Tree Learning

We proposed a novel architecture to learn temporal dependencies present in data streams. The proposal is based on the combination of the reservoir in the Echo State Network (ESN) and a Hoeffding Tree (on classification problems) or a FIMT-DD (regression).

We showed that the ESHT is able to learn three string-based functions typically implemented by a programmer. Our combination is able to learn faster than the standard ESN with fast adaptation to new unseen sequences as opposed to the standard FIMT-DD. The hyper-parameters required by the ESHT have a more predictable effect on the final accuracy than the hyper-parameters in typical neural networks (such as learning rate or momentum).

Despite the regression version of the ESHT achieving good results, the classification version did not performed as expected. On classification problems, we showed that our proposed architecture is able to improve a single HT on learning three well-known data streams datasets. However, it was not able to clearly outperform ensemble methods. In addition, there is no simple correlation between accuracy and hyper-parameters configuration on classification problems, which makes not very suitable for production environment as opposed to other well established methods such as the Adaptive Random Forest.

Resource-Aware Elastic Swap Adaptive Randomf Forest

The Elastic Swap Random Forest (ESRF) is an extension to the Adaptive Random Forest (ARF) ensemble method for reducing the number of base learners required. It extends ARF with two orthogonal components: 1) a swap component that splits learners into two sets based on their accuracy (only classifiers with the highest accuracy are used to make predictions); and 2) an elastic component for dynamically increasing or decreasing the number of classifiers in the ensemble.

We showed in the evaluations that how the two new components effectively contribute to reducing the number of classifiers up to one third (compared to the original ARF) while providing almost the same accuracy, resulting in speed-ups in terms of per-sample execution time close to 3x.

Furthermore, we made a sensitivity analysis of the two thresholds determining the elastic nature of the ensemble, establishing a trade–off in terms of resources (memory and computational requirements) and accuracy (which in all cases is comparable to the accuracy achieved by ARF100). This enables ESHT to be deployed on more constrained hardware environments, such those used in the IoT.

Ultra-low latency Random Forest

This work goes one big step further in fulfilling the low-latency requirements of today and future real-time analytics by exploiting current capabilities available on modern processors. We provide an flat ensemble architecture which is modular and adaptive to a variety of hardware platforms, from server to edge computing.

Accuracy of the proposed design has been validated with two reference implementations: MOA (for HT and Random Forest) and StreamDM (for HT). Throughput is evaluated on a variety of platforms. On Intel-based systems: i7 desktop (6 cores) and Xeon server (24 cores) class sockets. And on ARM-based systems: NVIDIA Jetson TX1 (4 cores), Applied Micro X-Gene2 (8 cores) and low-end Raspberry RPi3 (4 cores). For single HT the performance evaluation in terms of throughput reports 6.7x (i7), around 2x (Jetson TX1 and X-Gene2) and above 1x (RPi3) compared to MOA executed in i7. For Random Forest the evaluation reports throughput improvements of 85x (i7), 143x (Xeon parsing from memory), 10x (Jetson TX1) and 23x (X-Gene2) compared to single-threaded MOA on i7. The proposed multi-threaded implementation for the ensemble shows good scalability up to the largest core count socket that we have evaluated (75% parallel efficiency when using 24 cores on the Intel Xeon).

The evaluation also reports how the parser thread, in charge of feeding instances to the HT and ensemble, can easily limit throughput. The limits are mainly observed because of the media used to store the data (GPFS, solid–state disks, eMMC, SD, ...) that feeds the learners.

8.2 Future Work

Our future work research involves extending the contributions presented in chapters 6 and 7.

Resource-Aware Elastic Swap Random Forest

As part of our future work, we plan to improve the resize logic, trying to make it more adaptive and correlated with the performance evolution of each tree and their drift detectors. Also, we plan to make it more generic by extending the work to other ensemble architectures. Finally, we would like to evaluate our proposal with other base learners.

Ultra-low latency Random Forest

Future work involves investigating if the proposed single–parser design limits scalability and finding the optimal number of learners per parser ratio. How to improve the implementation in order to consider counters for attributes other than binary, is also part of our potential future work. We would like to work on scaling the evaluation to multi-socket nodes in which NUMA may be critical for performance. Finally, for dealing with large quantities of data, we would like to distribute the ensemble across several nodes in a cluster/distributed system.

Bibliography

[1] R. Agrawal, T. Imielinski, and A. Swami, *Database mining: A performance perspective*, IEEE Trans. on Knowl. and Data Eng., 5 (1993), pp. 914–925.

[2] G. H. R. K. Albert Bifet and B. Pfahringer, *Moa data stream mining: A practical approach*, 2011.

[3] P. Baldi, P. Sadowski, and D. O. Whiteson, *Searching for exotic particles in high-energy physics with deep learning.*, Nature communications, 5 (2014), p. 4308.

[4] J. P. Barddal, H. M. Gomes, and F. Enembreck, *Sfnclassifier: A scale-free social network method to handle concept drift*, in Proceedings of the 29th Annual ACM Symposium on Applied Computing, SAC '14, New York, NY, USA, 2014, ACM, pp. 786–791.

[5] G. E. A. P. A. Batista and M. C. Monard, *A study of k-nearest neighbour as an imputation method*, in HIS, 2002, pp. 251–260.

[6] Y. Ben-Haim and E. Tom-Tov, *A streaming parallel decision tree algorithm*, J. Mach. Learn. Res., 11 (2010), pp. 849–872.

[7] A. Besedin, P. Blanchart, M. Crucianu, and M. Ferecatu, *Evolutive deep models for online learning on data streams with no storage*, in ECML/PKDD 2017 Workshop on Large-scale Learning from Data Streams in Evolving Environments, Skopje, Macedonia, September 2017.

[8] A. Bifet, G. de Francisci Morales, J. Read, G. Holmes, and B. Pfahringer, *Efficient online evaluation of big data stream classifiers*, in Proceedings of the 21th ACM SIGKDD International Con-

ference on Knowledge Discovery and Data Mining, KDD '15, New York, NY, USA, 2015, ACM, pp. 59–68.

[9] A. Bifet, E. Frank, G. Holmes, and B. Pfahringer, *Ensembles of restricted hoeffding trees*, ACM Trans. Intell. Syst. Technol., 3 (2012), pp. 30:1–30:20.

[10] A. Bifet and R. Gavaldà, *Learning from Time-Changing Data with Adaptive Windowing*, in 2007 SIAM International Conference on Data Mining (SDM'07), 2007.

[11] A. Bifet and R. Gavaldà, *Adaptive learning from evolving data streams*, in Advances in Intelligent Data Analysis VIII, N. M. Adams, C. Robardet, A. Siebes, and J.-F. Boulicaut, eds., Berlin, Heidelberg, 2009, Springer Berlin Heidelberg, pp. 249–260.

[12] A. Bifet, G. Holmes, R. Kirkby, and B. Pfahringer, *MOA: Massive online analysis*, J. Mach. Learn. Res., 11 (2010), pp. 1601–1604.

[13] A. Bifet, G. Holmes, and B. Pfahringer, *Leveraging Bagging for Evolving Data Streams*, in Machine Learning and Knowledge Discovery in Databases, Springer, 2010, pp. 135–150–150.

[14] A. Bifet and R. Kirkby, *Data stream mining a practical approach*, (2009).

[15] J. A. Blackard and D. J. Dean, *Comparative accuracies of artificial neural networks and discriminant analysis in predicting forest cover types from cartographic variables*, 1999.

[16] L. K. Brandon S Parker, Ahmad Mustafa, *Novel class detection and feature via a tiered ensemble approach for stream mining*, 24th IEEE International Conference on Tools with Artificial Intelligence, (2012).

[17] L. Breiman, *Bagging predictors*, Machine Learning, 24 (1996), pp. 123–140.

[18] L. Breiman, *Random forests*, Machine Learning, 45 (2001), pp. 5–32.

[19] L. Breiman, J. Friedman, C. Stone, and R. Olshen, *Classification and Regression Trees*, The Wadsworth and Brooks-Cole statistics-probability series, Taylor & Francis, 1984.

[20] L. Breiman, D. Wolpert, P. Chan, and S. Stolfo, *Pasting small votes for classification in large databases and on-line*, in Machine Learning, 1999, pp. 85–103.

[21] T. Bujlow, T. Riaz, and J. M. Pedersen, *Classification of http traffic based on c5.0 machine learning algorithm*, in 2012 IEEE Symposium on Computers and Communications (ISCC), July 2012.

[22] R. Caruana and A. Niculescu-mizil, *An empirical comparison of supervised learning algorithms*, in In Proc. 23 rd Intl. Conf. Machine learning (ICML'06, 2006, pp. 161–168.

[23] A. Cauchy, *Méthode générale pour la résolution des systemes d'équations simultanées*, 1847, pp. 536–538.

[24] G. Cauwenberghs and T. Poggio, *Incremental and decremental support vector machine learning*, in Proceedings of the 13th International Conference on Neural Information Processing Systems, NIPS'00, Cambridge, MA, USA, 2000, MIT Press, pp. 388–394.

[25] K. Compatition, *Give me some credit dataset*.
https://www.kaggle.com/c/GiveMeSomeCredit.

[26] M. Datar, A. Gionis, P. Indyk, and R. Motwani, *Maintaining stream statistics over sliding windows: (extended abstract)*, in Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '02, Philadelphia, PA, USA, 2002, Society for Industrial and Applied Mathematics, pp. 635–644.

[27] A. P. Dawid, *Present position and potential developments: Some personal views: Statistical theory: The prequential approach*, 1984, pp. 278–292.

[28] R. Dey and F. M. Salem, *Gate-variants of gated recurrent unit (GRU) neural networks*, CoRR, abs/1701.05923 (2017).

[29] T. G. Dietterich, *Ensemble methods in machine learning*, in Proceedings of the First International Workshop on Multiple Classifier Systems, MCS '00, London, UK, UK, 2000, Springer-Verlag, pp. 1–15.

[30] P. Domingos and G. Hulten, *Mining high-speed data streams*, in Proceedings of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2000, pp. 71–80.

[31] P. Domingos and G. Hulten, *Mining high-speed data streams*, in Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '00, 2000, pp. 71–80.

[32] J. Duchi, E. Hazan, and Y. Singer, *Adaptive subgradient methods for online learning and stochastic optimization*, Tech. Rep. UCB/EECS-2010-24, EECS Department, University of California, Berkeley, Mar 2010.

[33] E. Frank, M. A. Hall, and I. H. Witten, *The WEKA Workbench. Online Appendix for "Data Mining: Practical Machine Learning Tools and Techniques"*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4th ed., 2016.

[34] E. Frank, G. Holmes, R. Kirkby, and M. Hall, *Racing committees for large datasets*, in Discovery Science, S. Lange, K. Satoh, and C. H.

Smith, eds., Berlin, Heidelberg, 2002, Springer Berlin Heidelberg, pp. 153–164.

[35] Y. Freund and R. E. Schapire, *A decision-theoretic generalization of on-line learning and an application to boosting*, 1996.

[36] Y. Freund and R. E. Schapire, *A decision-theoretic generalization of on-line learning and an application to boosting*, J. Comput. Syst. Sci., 55 (1997), pp. 119–139.

[37] J. Gama, I. Žliobaitė, A. Bifet, M. Pechenizkiy, and A. Bouchachia, *A survey on concept drift adaptation*, ACM Comput. Surv., 46 (2014), pp. 44:1–44:37.

[38] J. a. Gama, R. Rocha, and P. Medas, *Accurate decision trees for mining high-speed data streams*, in Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '03, New York, NY, USA, 2003, ACM, pp. 523–528.

[39] A. Ghazikhani, R. Monsefi, and H. sadoghi yazdi, *Online neural network model for non-stationary and imbalanced data stream classification*, International Journal of Machine Learning and Cybernetics, 5 (2014), pp. 51–62.

[40] Github, ed., *StreamDM-C++: C++ Stream Data Mining*, IEEE Computer Society, 2015.

[41] C. Goller and A. Küchler, *Learning task-dependent distributed representations by backpropagation through structure*, Proceedings of International Conference on Neural Networks (ICNN'96), 1 (1996), pp. 347–352 vol.1.

[42] H. M. Gomes, J. P. Barddal, L. E. Boiko, and A. Bifet, *Adaptive random forests for data stream regression*, April 2018.

[43] H. M. Gomes, J. P. Barddal, F. Enembreck, and A. Bifet, *A survey on ensemble learning for data stream classification*, ACM Comput. Surv., 50 (2017), pp. 23:1–23:36.

[44] H. M. Gomes, A. Bifet, J. Read, J. P. Barddal, F. Enembreck, B. Pfharinger, G. Holmes, and T. Abdessalem, *Adaptive random forests for evolving data stream classification*, Machine Learning, 106 (2017), pp. 1469–1495.

[45] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, *Generative adversarial nets*, in Advances in Neural Information Processing Systems 27, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, eds., Curran Associates, Inc., 2014, pp. 2672–2680.

[46] B. Gu, V. S. Sheng, Z. Wang, D. Ho, S. Osman, and S. Li, *Incremental learning for v-support vector regression*, Neural Networks, 67 (2015), pp. 140 – 150.

[47] J. Hadamard, *Théorie des équations aux dérivées partielles linéaires hyperboliques et du problème de cauchy*, Acta Math., 31 (1908), pp. 333–380.

[48] M. Harries, U. N. cse tr, and N. S. Wales, *Splice-2 comparative evaluation: Electricity pricing*, tech. rep., 1999.

[49] S. Hashemi, Y. Yang, Z. Mirzamomen, and M. Kangavari, *Adapted one-vs-all decision trees for data stream classification*, 2009.

[50] K. He, X. Zhang, S. Ren, and J. Sun, *Delving deep into rectifiers: Surpassing human-level performance on imagenet classification*, in The IEEE International Conference on Computer Vision (ICCV), December 2015.

[51] R. Hecht-Nielsen, *Neural networks for perception (vol. 2)*, Harcourt Brace & Co., Orlando, FL, USA, 1992, ch. Theory of the Backpropagation Neural Network, pp. 65–93.

[52] J. Heck and F. M. Salem, *Simplified minimal gated unit variations for recurrent neural networks*, CoRR, abs/1701.03452 (2017).

[53] S. Hochreiter and J. Schmidhuber, *Long short-term memory*, Neural Comput., 9 (1997), pp. 1735–1780.

[54] W. Hoeffding, *Probability inequalities for sums of bounded random variables*, Journal of the American Statistical Association, 58 (1963), pp. 13–30.

[55] G. Holmes, R. Kirkby, and B. Pfahringer, *Stress-testing hoeffding trees*, in Knowledge Discovery in Databases: PKDD 2005, A. M. Jorge, L. Torgo, P. Brazdil, R. Camacho, and J. Gama, eds., Berlin, Heidelberg, 2005, Springer Berlin Heidelberg, pp. 495–502.

[56] G. Huang, *What are extreme learning machines? filling the gap between frank rosenblatt's dream and john von neumann's puzzle*, Cognitive Computation, 7 (2015), pp. 263–278.

[57] G.-B. Huang, L. Chen, and C.-K. Siew, *Universal approximation using incremental constructive feedforward networks with random hidden nodes*, Neural Networks, IEEE Transactions on, 17 (2006), pp. 879–892.

[58] G. Hulten, L. Spencer, and P. Domingos, *Mining time-changing data streams*, in Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '01, New York, NY, USA, 2001, ACM, pp. 97–106.

[59] G. Hulten, L. Spencer, and P. Domingos, *Mining time-changing data streams*, in Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '01, New York, NY, USA, 2001, ACM, pp. 97–106.

[60] G. Hulten, L. Spencer, and P. Domingos, *Mining time-changing data streams*, in Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '01, ACM, 2001, pp. 97–106.

[61] E. Ikonomovska, *Airline dataset for classification*. http://kt.ijs.si/elena_ikonomovska/data.html.

[62] E. Ikonomovska, J. Gama, and S. Džeroski, *Learning model trees from evolving data streams*, Data Mining and Knowledge Discovery, 23 (2010), pp. 128–168.

[63] Intel, *Optimizing performance with intel advanced vector extensions. intel white paper*, 2014.

[64] H. Jaeger, *The "echo state" approach to analysing and training recurrent neural networks - with an erratum note*, tech. rep., German National Research Center for Information Technology, 2001.

[65] P. Kang, *Locally linear reconstruction based missing value imputation for supervised learning*, Neurocomputing, 118 (2013), pp. 65–78.

[66] M. Khan, Q. Ding, and W. Perrizo, *K-nearest neighbor classification on spatial data streams using p-trees*, 12 2001.

[67] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, in ICLR, 2015.

[68] J. Z. Kolter and M. A. Maloof, *Dynamic weighted majority: A new ensemble method for tracking concept drift*, in Proceedings of the Third IEEE International Conference on Data Mining, ICDM '03, 2003, pp. 123–130.

[69] N. Kourtellis, G. D. F. Morales, A. Bifet, and A. Murdopo, *Vht: Vertical hoeffding tree*, in 2016 IEEE International Conference on Big Data (Big Data), Dec 2016, pp. 915–922.

[70] L. I. Kuncheva and C. J. Whitaker, *Measures of diversity in classifier ensembles and their relationship with the ensemble accuracy*, Machine Learning, 51 (2003), pp. 181–207.

[71] K. Lang, *20 newsgroups data set*, 2008. Last accessed: October 2016.

[72] Y.-N. Law and C. Zaniolo, *An adaptive nearest neighbor classification algorithm for data streams*, in Knowledge Discovery in Databases: PKDD 2005, A. M. Jorge, L. Torgo, P. Brazdil, R. Camacho, and

J. Gama, eds., Berlin, Heidelberg, 2005, Springer Berlin Heidelberg, pp. 108–120.

[73] W. Ling and F. Dong-Mei, *Estimation of missing values using a weighted k-nearest neighbors algorithm*, Environmental Science and Information Application Technology, International Conference on, 3 (2009), pp. 660–663.

[74] V. Losing, B. Hammer, and H. Wersing, *KNN Classifier with Self Adjusting Memory for Heterogeneous Concept Drift*, in 2016 IEEE 16th International Conference on Data Mining (ICDM), IEEE, 2016, pp. 291–300.

[75] M. Lukoševičius, *A Practical Guide to Applying Echo State Networks*, in Neural Networks: Tricks of the Trade, vol. 7700 of LNCS, Springer Berlin Heidelberg, 2012, ch. 27, pp. 659–686.

[76] M. Lukoševičius and H. Jaeger, *Survey: Reservoir computing approaches to recurrent neural network training*, Comput. Sci. Rev., 3 (2009), pp. 127–149.

[77] W. Maass and H. Markram, *On the computational power of recurrent circuits of spiking neurons*, Electronic Colloquium on Computational Complexity (ECCC), (2002).

[78] W. Maass, T. Natschläger, and H. Markram, *Real-time computing without stable states: A new framework for neural computation based on perturbations*, Neural Comput., 14 (2002), pp. 2531–2560.

[79] D. Marron, G. D. F. Morales, and A. Bifet, *Random forests of very fast decision trees on gpu for mining evolving big data streams*, in Proceedings of ECAI 2014, 2014.

[80] D. Marrón, J. Read, A. Bifet, and N. Navarro, *Data stream classification using random feature functions and novel method combinations*, Journal of Systems and Software, (2016).

[81] J. Martens and I. Sutskever, *Learning recurrent neural networks with hessian-free optimization*, in Proceedings of the 28th International Conference on Machine Learning, ICML 2011, Bellevue, Washington, USA, June 28 - July 2, 2011, 2011, pp. 1033–1040.

[82] R. Martin, *Agile Software Development: Principles, Patterns, and Practices*, Alan Apt series, Pearson Education, 2003.

[83] L. L. Minku, A. P. White, and X. Yao, *The impact of diversity on online ensemble learning in the presence of concept drift*, IEEE Trans. on Knowl. and Data Eng., 22 (2010), pp. 730–742.

[84] L. L. Minku and X. Yao, *Ddd: A new ensemble approach for dealing with concept drift*, IEEE Transactions on Knowledge and Data Engineering, 24 (2012), pp. 619–633.

[85] H. Mouss, D. Mouss, N. Mouss, and L. Sefouhi, *Test of page-hinckley, an approach for fault detection in an agro-alimentary production system*, in 2004 5th Asian Control Conference (IEEE Cat. No.04EX904), vol. 2, July 2004, pp. 815–818 Vol.2.

[86] N. C. Oza, *Online bagging and boosting*, 2005 IEEE International Conference on Systems, Man and Cybernetics, 3 (2001), pp. 2340–2345 Vol. 3.

[87] M. C. Ozturk and J. C. Príncipe, *An associative memory readout for ESNs with applications to dynamical pattern recognition.*, Neural Networks, 20 (2007), pp. 377–390.

[88] R. Polikar, *Ensemble based systems in decision making*, IEEE Circuits and Systems Magazine, 6 (2006), pp. 21–45.

[89] R. Polikar, L. Upda, S. S. Upda, and V. Honavar, *Learn++: an incremental learning algorithm for supervised neural networks*, IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews), 31 (2001), pp. 497–508.

[90] M. Pratama, P. Angelov, j. lu, E. Lughofer, m. seera, and c. peng lim, *A randomized neural network for data streams*, 05 2017.

[91] N. Qian, *On the momentum term in gradient descent learning algorithms*, Neural Netw., 12 (1999), pp. 145–151.

[92] W. Qu, Y. Zhang, J. Zhu, and Q. Qiu, *Mining multi-label concept-drifting data streams using dynamic classifier ensemble*, in Asian Conference on Machine Learning, vol. 5828 of Lecture Notes in Computer Science, Springer, 2009, pp. 308–321.

[93] J. Read, A. Bifet, B. Pfahringer, and G. Holmes, *Batch-incremental versus instance-incremental learning in dynamic and evolving data*, in Advances in Intelligent Data Analysis XI, J. Hollmén, F. Klawonn, and A. Tucker, eds., Berlin, Heidelberg, 2012, Springer Berlin Heidelberg, pp. 313–323.

[94] L. Rokach, *Ensemble-based classifiers*, Artificial Intelligence Review, 33 (2010), pp. 1–39.

[95] M. Roseberry and A. Cano, *Multi-label knn classifier with self adjusting memory for drifting data streams*, in Proceedings of the Second International Workshop on Learning with Imbalanced Domains: Theory and Applications, Proceedings of Machine Learning Research, ECML-PKDD, Dublin, Ireland, 10 Sep 2018, PMLR, pp. 23–37.

[96] D. E. Rumelhart, J. L. McClelland, and C. PDP Research Group, eds., *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1: Foundations*, MIT Press, Cambridge, MA, USA, 1986.

[97] D. Sahoo, Q. Pham, J. Lu, and S. C. H. Hoi, *Online deep learning: Learning deep neural networks on the fly*, in Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18, International Joint Conferences on Artificial Intelligence Organization, 7 2018, pp. 2660–2666.

[98] A. Salazar, G. Safont, A. Soriano, and L. Vergara, *Automatic credit card fraud detection based on non-linear signal processing*, in 2012 IEEE International Carnahan Conference on Security Technology (ICCST), Oct 2012.

[99] B. Schrauwen, D. Verstraeten, and J. V. Campenhout, *An overview of reservoir computing: theory, applications and implementations*, in Proceedings of the 15th European Symposium on Artificial Neural Networks, 2007, pp. 471–482.

[100] A. Shaker and E. Hüllermeier, *Instance-based classification and regression on data streams*, in Learning in Non-Stationary Environments, Springer New York, 2012, pp. 185–201.

[101] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, *Mastering the game of go with deep neural networks and tree search*, Nature, 529 (2016), pp. 484–489.

[102] W. N. Street and Y. Kim, *A streaming ensemble algorithm (sea) for large-scale classification*, in Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '01, 2001, pp. 377–382.

[103] M. Tennant, F. Stahl, O. Rana, and J. B. Gomes, *Scalable real-time classification of data streams with concept drift*, Future Generation Computer Systems, 75 (2017), pp. 187 – 199.

[104] M. Tennant, F. Stahl, O. Rana, and J. B. Gomes, *Scalable real-time classification of data streams with concept drift*, Future Generation Computer Systems, (2017).

[105] M. Thompson, D. Farley, M. Barker, P. Gee, and A. Stewart, *DISRUPTOR: High performance alternative to bounded queues for exchanging data between concurrent threads*, 2015.

[106] T. Tieleman and G. Hinton, *Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude*, tech. rep., 2012.

[107] S. Tulyakov, S. Jaeger, V. Govindaraju, and D. Doermann, *Review of Classifier Combination Methods*, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 361–386.

[108] P. E. Utgoff, *Incremental induction of decision trees*, Machine Learning, 4 (1989), pp. 161–186.

[109] P. E. Utgoff, N. C. Berkman, and J. A. Clouse, *Decision tree induction based on efficient tree restructuring*, Machine Learning, 29 (1997), pp. 5–44.

[110] D. S. Vogel, O. Asparouhov, and T. Scheffer, *Scalable look-ahead linear regression trees*, in Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '07, New York, NY, USA, 2007, ACM, pp. 757–764.

[111] I. Žliobaitė, *Combining time and space similarity for small size learning under concept drift*, in Foundations of Intelligent Systems, J. Rauch, Z. W. Raś, P. Berka, and T. Elomaa, eds., Berlin, Heidelberg, 2009, Springer Berlin Heidelberg, pp. 412–421.

[112] I. Žliobaitė, *Adaptive training set formation*, PhD thesis, Vilnius University, 2010.

[113] P. J. Werbos, *Generalization of backpropagation with application to a recurrent gas market model*, Neural Networks, (1988), pp. 339 – 356.

[114] I. B. Yildiz, H. Jaeger, and S. J. Kiebel, *Re-visiting the echo state property*, Neural Networks, 35 (2012), pp. 1 – 9.

[115] L. Zhang and P. Suganthan, *A survey of randomized algorithms for training neural networks*, Inf. Sci., 364 (2016), pp. 146–155.

[116] P. Zhang, B. J. Gao, X. Zhu, and L. Guo, *Enabling fast lazy learning for data streams*, in 2011 IEEE 11th International Conference on Data Mining, Dec 2011, pp. 932–941.