Universidade Federal de São Carlos– UFSCar
Centro de Ciências Exatas e de Tecnologia– CCET
Departamento de Computação– DC
Programa de Pós-Graduação em Ciência da Computação– PPGCC

Guilherme Weigert Cassales

Improving data locality of bagging ensembles for data streams through mini-batching

São Carlos
2021

Guilherme Weigert Cassales

Improving data locality of bagging ensembles for data streams through mini-batching

Tese apresentada ao Programa de Pós-Graduação em Ciência da Computação do Centro de Ciências Exatas e de Tecnologia da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Doutor em Ciência da Computação.

Área de concentração: Sistemas de Computação

Supervisor: Hermes Senger

São Carlos
2021

UNIVERSIDADE FEDERAL DE SÃO CARLOS

Centro de Ciências Exatas e de Tecnologia
Programa de Pós-Graduação em Ciência da Computação

Folha de Aprovação

Defesa de Tese de Doutorado do candidato Guilherme Weigert Cassales, realizada em 27/07/2021.

Comissão Julgadora:

Prof. Dr. Hermes Senger (UFSCar)

Prof. Dr. Murilo Coelho Naldi (UFSCar)

Prof. Dr. Bernhard Pfahringer (UW)

Prof. Dr. Samuel Xavier de Souza (UFRN)

Prof. Dr. Marco Aurélio Stelmar Netto (IBM Research)

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001.

Acknowledgements

I want to thank everyone that contributed in any way to the execution of this thesis.

Firstly, I would like to thank my supervisor Hermes Senger. Even though the road was quite bumpy, he never lost hope that eventually, something interesting would come out at the end of this journey. In the meantime, we forged an excellent bond that should keep bearing fruits in the future.

I would like to thank the Programa de Pós-graduação em Ciência da Computação (PPGCC) from the Universidade Federal de São Carlos (UFSCar) and the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001, that provided the fundings to allow the development of this work.

Also, I am forever thankful to the Programa Institucional de Internacionalização CAPES-PrInt UFSCar (Contract 88887.373234/2019-00) for providing financial support to my internship in New Zealand and the Machine Learning group of the University of Waikato, especially Prof. Albert Bifet, Prof. Bernhard Pfrahringer, and Dr. Heitor Gomes for receiving me and providing critical feedback to my work during this period.

In addition, I would like to thank Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP - contract numbers 2018/22979-2) and Prof. Daniel Macêdo Batista for providing support during the last stages of this work.

Lastly, I would like to thank my family and friends who helped me by providing emotional support and encouragement during my studies. Among the close friends not familiar with academia, I would like to give special thanks to those who did not commit any of the following sins:

1. Ask me how my research was going as I approached the deadline;
2. Ask me, “do you work as well or do you just study?”;
3. Ask me, “it is Saturday. Why are you working until 03 am?”.

Resumo

As técnicas de aprendizado de máquina foram empregadas em praticamente todos os domínios nos últimos anos. Em muitas aplicações, os algoritmos de aprendizagem terão que lidar com ambientes dinâmicos, onde precisam fornecer uma resposta em (quase)tempo real enquanto aderem com restrições tanto de memória quanto de tempo. Nesse cenário, os comitês de aprendizagem compreendem uma classe de algoritmos de mineração de fluxo de dados capaz de alcançar um notável desempenho preditivo. Os comitês de aprendizagem são implementados como um conjunto de (vários) classificadores individuais, cujas predições são agregadas para classificar novas instâncias de entrada. Embora os comitês de aprendizagem possam ser computacionalmente mais caros, eles são naturalmente modificáveis para o paralelismo de tarefas. No entanto, o aprendizado incremental e as estruturas de dados dinâmicas usadas para capturar o desvio de conceito aumentam as falhas de cache e podem reduzir o benefício do paralelismo.

Nesta tese, um método capaz de reduzir o tempo de execução e aumentar a eficiência energética de diversos comitês de aprendizagem do tipo bagging para fluxos de dados é proposto. O método é baseado em um modelo de paralelismo de tarefas capaz de aproveitar a independência natural dos classificadores internos que compõem os comitês de aprendizagem da classe bagging. O modelo paralelo é combinado com uma técnica de mini-batching, a qual pode melhorar a localidade de acesso à memória dos comitês de aprendizagem.

Speedups de 4X a 5X são alcançados consistentemente com 8 núcleos de processamento, apresentando, inclusive, um Speedup superlinear de 12X em um caso específico. Demonstra-se que o mini-batching pode diminuir significativamente a distância de reuso e o número de falhas de cache. Fornece-se dados sobre a relação de compromisso da redução do tempo de execução com a perda no desempenho preditivo, perda que pode variar de menos de 1% a até 12%. Conclui-se que a perda de desempenho preditivo depende, principalmente, das características do conjunto de dados e do tamanho do mini-batch utilizado. Apresenta-se evidências de que o uso de mini-batches pequenos (por exemplo, até

50 exemplos) fornece um bom compromisso entre o tempo de execução e o desempenho preditivo.

Demonstra-se que a eficiência energética pode ser melhorada em utilizando três níveis de carga de trabalho diferentes. Embora a maior redução no consumo de energia aconteça com o menor nível de carga de trabalho a contrapartida é um grande atraso no tempo de resposta, o que pode atrapalhar a ideia de processamento em tempo real. Nos níveis de cargas de trabalho mais altos, entretanto, o método proposto apresenta melhor desempenho tanto no consumo de energia quanto no atraso no tempo de resposta quando comparado à versão base.

Avalia-se o método utilizando diversas plataformas de hardware, com um total de seis plataformas diferentes utilizadas entre todos os frameworks experimentais. Ao mesmo tempo, utiliza-se de até seis algoritmos diferentes e até cinco conjuntos de dados diferentes nos frameworks experimentais.

Ao fornecer dados sobre a execução do método proposto em uma gama tão ampla de configurações, acredita-se que o método proposto é uma solução viável para melhorar o desempenho de comitês de aprendizagem da classe bagging para fluxos de dados.

Palavras-chave: Aprendizagem de fluxo de dados. Paralelismo de tarefas multicore. Comitês de aprendizagem. Algoritmos de bagging. Múltiplas plataformas. Consumo de energia..

Abstract

Machine Learning techniques have been employed in virtually all domains in the past few years. In many applications, learning algorithms will have to cope with dynamic environments, under both memory and time constraints, to provide a (near) real-time answer. In this scenario, Ensemble learning comprises a class of stream mining algorithms that achieved remarkable predictive performance. Ensembles are implemented as a set of (several) individual learners whose predictions are aggregated to predict new incoming instances. Although ensembles can be computationally more expensive, they are naturally amendable for task-parallelism. However, the incremental learning and dynamic data structures used to capture the concept drift increase the cache misses and hinder the benefit of parallelism.

In this thesis, we devise a method capable of reducing the execution time and increasing the energy efficiency of several bagging ensembles for data streams. The method is based on a task-parallel model capable of leveraging the natural independence of the underlying learners from this class of ensembles (bagging). The parallel model is combined with a mini-batching technique that can improve the memory access locality of the ensembles.

We consistently achieve speedups of 4X to 5X with 8 cores, with even a superlinear speedup of 12X in one case. We demonstrate that mini-batching can significantly decrease the reuse distance and the number of cache-misses. We provide data regarding the trade-off regarding the reduction of execution time with a loss in predictive performance (ranging from less than 1% up to 12%). We conclude that loss in predictive performance depends on dataset characteristics and the mini-batch size used. We present evidence that using small mini-batch sizes (e.g., up to 50 examples) provides a good compromise between execution time and predictive performance.

We demonstrate that energy efficiency can be improved under three different workloads. Although the biggest reduction in energy consumption happens in the smallest workload, it comes at the cost of a big delay in response time, which may hinder the idea of real-time processing. In the higher workloads, however, the proposed method presents a better performance in both the energy consumption and the delay in response time when compared to the baseline version.

We evaluate our method using many hardware platforms, with a total of six different hardware platforms used among all experimental frameworks. At the same time, we use up to six different algorithms and up to five different datasets on the experimental frameworks.

By providing data about the execution of the proposed method in such a wide range of setups, we believe that the proposed method is a viable solution for improving the performance of online bagging ensembles.

Keywords: Data stream learning. Multicore task-parallelism. Ensemble learners. Bagging algorithms. Multiple platforms. Energy consumption. .

List of Figures

List of Tables

Contents

Chapter 1

Introduction

Nowadays, digital devices are getting, in general, both more affordable and more powerful compared to the beginning of the century. Besides growing in the number of devices, our digital world is also getting smarter. In today's world, advertisements are "custom made", and platforms recommend content aligned to the user's taste. Surprisingly or not, these services only need the user's history of actions on their systems to provide a better experience. The moving force behind such power is Machine Learning (ML), which has become fundamental for many applications in different domains.

Historically, ML algorithms worked with the idea of creating models for static datasets, also called batch learning, in which the entire dataset is available for the training phase (GAMA, 2010). This approach showed, in most cases, excellent results when making predictions on new data from the same source and data distribution.

Recent advances in hardware and software allowed the large scale acquisition of data (AGGARWAL, 2007). This data is usually depicted as an infinite sequence, generated continuously, usually at high speed, and is known as data streams (GAMA, 2010; GAMA; GABER, 2007). Many applications are deployed in these dynamic environments, where they have to cope with the specific challenges from the dynamic environments. At the same time, applications have to adhere to constraints such as using small and finite resources, detecting and reacting to changes in the predicted data, and providing the prediction in a short interval.

Due to the aforementioned constraints, many of the established solutions designed for the static setting are no longer effective when deployed in dynamic environments. In this context, methods that possessed remarkable predictive performance in the static setting have posed many challenges in achieving the same level of performance after adapting the algorithm to a dynamic setting.

Such is the case of the Bootstrap Aggregating (Bagging) proposed by (BREIMAN, 1996) in which many weak learners compose an ensemble. Although the base learners of an ensemble are individually weak and unstable, they can provide a more reliable prediction by combining the individual outputs as long as training of weak learners is performed independently (i.e., with different subsets of a dataset). The trade-off for better predictive performance is a higher demand of computational resources in the form of storage space, processing time, and computational power to process all the data. Even using weak learners, it may be challenging to use ensembles in dynamic environments since they are more computationally intensive and may violate the time and resource constraints. In addition, as data streams present transient behavior, prediction models often need to be incremented to adapt to concept drift observed in data. According to (GAMA; RODRIGUES, 2009), it is precisely due to the transient nature that keeping the decision model updated and maintaining a good level of accuracy is the big issue in learning from data streams.

Although parallelism can be a good strategy for performance improvement, many algorithms typically require collective communication with high overhead (EKANAYAKE et al., 2016). The algorithms derived from Bagging are an exception to this. Bagging ensembles do not depend on collective communication because each weak learner is trained independently, needing only a synchronization in the form of aggregating the votes from each learner. This characteristic may allow such ensembles to be parallelized while also adhering to memory and time constraints. However, ensemble models implement different data structures and are not amendable for data parallelism (GOMES et al., 2017a). In fact, task parallelism is the natural way to implement them because such ensembles operate on dynamic data structures used to model concept drift and non-stationary data behavior. In this scenario, memory access patterns and cache memory performance become one major challenge for the parallel implementation in multi-core environments.

1.1 Problem statement

It is easy to notice the need for a method to accelerate more complex ML models and enable the deployment of models with better predictive performance in dynamic environments with response time constraints. For many years, improvements in computer technology have been driven by Moore's Law (MOORE, 2006). During this period, computer performance could be improved simply by increasing the “clock cycle” of the processor, which directly reduces the time required to perform a basic operation. However, clock cycle times are decreasing slowly and appear to be approaching physical limits (FOSTER, 1995). Since we can no longer depend on the increasing clock cycle to process more data in less time, the alternative to this problem has been to incorporate multiple processors and perform the computations in parallel. This shift has been going on for a while, which translates to newer chips having more physical processor counts.

There are, however, multiple challenges awaiting those that wish to extract the best performance when operating with more computing units. When analyzing the specific problem of parallelization of ensemble learning methods, it is possible to note that, several papers focused on batch methods (YAN et al., 2009; BASILICO et al., 2011; PANDA et al., 2009; JAHNKE, 2009; XAVIER; THIRUNAVUKARASU, 2017; LIU, 2014; GHOTING et al., 2011) even though the idea is not new. Such works used MapReduce frameworks, which are not suitable for applications with requirements of low response times, even being capable of processing huge amounts of data with high scalability (SENGER et al., 2016). In the context of multi-core parallelism, there are instances of batch ML (HAJEWSKI; OLIVEIRA, 2020; ISLAM et al., 2009; CYGANEK; SOCHA, 2014; VALLE et al., 2010; HOYOS-IDROBO et al., 2018; NOJIMA; MIHARA; ISHIBUCHI, 2010; WEILL et al., 2019; HUSSAIN et al., 2012; JIN; AGRAWAL, 2003a). However, a few studies have already begun to explore data stream methods (HORAK; BERKA; VAJTERSIC, 2013; MARTINOVIC et al., 2019; QIAN et al., 2016; MARRóN et al., 2017).

Although several parallel ensemble algorithms have been proposed, methods focusing on their efficient implementation are seldom approached in the literature. In addition, methods may focus on only one algorithm, which may render their contribution hard to be reused. In this context, a strategy applicable to a wide array of algorithms and capable of being used in conjunction with individual optimizations may contribute to the state-of-the-art.

1.2 Hypothesis

This thesis is based on the following Hypothesis:

- ❏ **H1:** *Task-parallelism per se is sufficient to improve the performance of bagging ensembles for data streams.*

- ❏ **H2:** *Memory access patterns are a major bottleneck of parallel implementations. Improving the access pattern can lead to better computational performance.*

- ❏ **H3:** *It is possible to accelerate the execution time of bagging ensembles for data streams without significantly impacting (reducing) the predictive performance of the algorithms.*

1.3 Contributions

This thesis investigates ways to improve the performance of bagging ensembles used in data stream classification. We make the following contributions:

- ❏ We design and implement a task-parallelization model for data stream bagging ensembles by extending algorithms from a popular ML framework using 2 Java parallel APIs. We evaluate both APIs on two different platforms, using four algorithms and four datasets;
- ❏ We perform an exploratory study of the parallel implementations' efficiency, investigate the bottlenecks, and present data showing the major shortcomings of the initially proposed strategy. In addition, we conclude by showing which API is better in different situations;
- ❏ We evaluate a *mini-batching* technique that groups data instances and rearranges the order of operations in the bagging ensembles for data streams. This technique is capable of improving the memory access patterns of the algorithms.
- ❏ We demonstrated that the parallelization *per se* does not guarantee good performance. However, when augmented by a *mini-batching* technique capable of improving both the execution time and the energy efficiency of bagging ensembles.

During this work, we produced the following publications. In (CASSALES et al., 2019), we published a preliminary study where we experimented with three data-stream algorithms in a constrained device. This work was important as the ensemble showed a better predictive performance than the other two algorithms used. However, ensemble algorithms presented the slowest execution time, pointing to a need for acceleration of execution. In (CASSALES et al., 2020), we present the parallel mini-batching technique. In (CASSALES et al., 2021), we extend HPCC work with additional datasets and algorithms and the theory of data locality. Another article (under evaluation at the time of this writing) with additional results on how *mini-batching* improves the energy efficiency of bagging ensembles was submitted to a scientific journal.

During my Doctorate, I did an international internship, where I worked with high-quality researchers. I received a grant to stay six months working at the University of Waikato in New Zealand. In the internship, I worked with the Machine Learning group. During my internship, the main topic of this thesis was 'discovered', and I started working with it.

A direct collaboration of this thesis was published in (PUHL et al., in press), where the author implements the distributed architecture, proposed in the ISCC paper, using MPI. Finally, the present thesis is undergoing the registration of Intelectual Property.

1.4 Document structure

The remaining content of this thesis is organized as follows. A brief review of the main aspects and definitions regarding data stream classification, in the context of this thesis, is presented in Chapter 2. The first part of this chapter is used to define some key concepts for this thesis, such as what is a data stream, how classification for data streams differ from *batch* classification, and what are the constraints that the learner should operate under in a data stream scenario. In the second part of this chapter, ensemble methods for data stream classification are presented.

Chapter 3 provides the a brief review of the main aspects and definition regarding Parallel Processing applied to Machine Learning, in the context of this thesis. A review of the standard parallelization process is provided, followed by a summary of the frameworks and tools available and a discussion on data-locality, a key aspect of efficient implementation.

Chapter 4 presents the related literature, highlighting the main aspects of related works and the key gaps addressed in this thesis.

Chapter 5 presents a parallel model to data stream bagging ensembles and an experimental evaluation of such model. We present several assumptions and datasets used throughout this thesis. We present a methodology for the evaluation as well as the results analysis. We finish the development of this chapter with the report of an investigation made to find out the bottlenecks of the implementation. Finally, we conclude with a summary and closing thoughts of the chapter, pointing to the next developments.

Chapter 6 introduces a technique called *mini-batching* to data stream bagging ensembles. We show how this technique changes execution times of sequential bagging ensembles. We present theoretical evidence, based on the data-locality material from section 3.3, that it is an optimal solution. We also evaluate *mini-batching* impact on the predictive performance. We conclude with a summary and closing thoughts of the chapter, pointing to the next developments.

Chapter 7 shows the combination of the parallelization model and the *mini-batching* technique. We present experimental evaluation to show the improvements in performance. We provide real data regarding memory access that sustains the claims made about the *mini-batching* technique. We conclude with a summary and closing thoughts of the chapter, pointing to the next developments.

Chapter 8 presents an energy-efficiency study using the combined methods. After a brief introduction to the energy problem, we present an experimental evaluation to show that our method is capable of improving the energy efficiency of the algorithms under several workloads. We conclude with a summary and closing thoughts of the chapter, pointing to future work possibilities.

Finally, Chapter 9 provides a summary of the contributions of this thesis, reflects on their implications, and speculates about future research directions that could follow the work that has been undertaken.

Chapter 2

Data stream mining

Recent advances in hardware and software allowed the large-scale acquisition of data (AGGARWAL, 2007). However, data, by itself, provides a minimal competitive advantage and must be processed in order to extract information that can create value and aid in decision making. The volume of data generated by applications like social networks and sensors of wide variety has been growing every year, which increases the demand for fast and efficient solutions capable of processing it in the shortest interval possible. Recently, the community has been focusing its efforts to obtain valuable models from massive amounts of rapidly generated data, giving birth to the area of data stream mining (GOMES et al., 2017a).

Formally, a data stream S is a massive sequence of data elements $x_1, x_2, \ldots, x_n$ that is, $S = \{x_i\}_{i=1}^{n}$, which is potentially unbounded (i.e., n $\rightarrow \infty$) and arrives at high speeds (SILVA et al., 2013).

This chapter defines some key concepts and challenges about the data stream classification (section 2.1) and, more specifically, the data stream ensemble algorithms (section 2.2) used in developing this thesis.

2.1 Data stream classification

The objective of classification in data streams is to predict, with high accuracy, the class of unknown examples that are arriving continuously in the form of a data stream. Data stream classification is a sequence of examples, where each example is a set of d attributes (CHEN; ZOU; TU, 2009). Each attribute can be either numeric or nominal. Each example is associated with a label, represented by a discrete value that indicates the class that the example belongs.

Except that data arrives in the form of a data stream, this definition may seem very similar to the traditional classification problem. However, data streams differ from traditional data in many ways, as defined by (BABCOCK et al., 2002):

- ❏ Data arrives continuously, and usually, in high speed;
- ❏ The system has no control over the order in which the examples arrive to be processed;
- ❏ It is impossible to store all data examples;
- ❏ Data streams are possibly infinite. In less extreme cases, they are at least many times bigger than the physical memory;
- ❏ Once processed, the example has to be discarded or summarized. Example recovery is a challenging task;

In addition, data stream classification is usually deployed in environments with a dynamic nature, where the data distribution is non-stationary and may change over time. This phenomenon is known as concept drift (TSYMBAL, 2004). According to (GAMA; RODRIGUES, 2009), the biggest challenge in data stream classification is keeping an up-to-date model with high accuracy, which requires incremental learning algorithms because these algorithms consider the concept drift associated with the process. In this case, the model might need to discard old examples – which may not reflect the present data distribution – and adapt the decision model to the new data.

Even though concept drift has been extensively researched, most works assume data distribution is independent, meaning that the label of an example does not depend on the previous examples of the stream. This dependency is prevalent in data streams coming from data recording devices like video surveillance and sensors. Therefore, temporal dependence adds yet another challenge to data stream classification (ŽLIOBAITė et al., 2014).

In summary, data stream poses several challenges for learning algorithms, including, but not limited to: a massive amount of examples, high arrival speeds, limited labeled examples, restricted resources (time and memory), novel classes, concept and feature drifts, a trade-off between accuracy and efficiency, distributed applications, and temporal dependencies (GOMES et al., 2017a; AGGARWAL, 2007; BARDDAL et al., 2016; KRAWCZYK et al., 2017a)

2.1.1 Hoeffding Tree

One of the earliest algorithms to deal with these challenges was the Hoeffding Tree (DOMINGOS; HULTEN, 2000), also known as Very Fast Decision Tree (VFDT). The Hoeffding Tree (HT) is an incremental tree designed to cope with massive data streams.

Thus, it can create splits with reasonable confidence in the data distribution while having very few instances available. This is possible because of the Hoeffding Bound (HB), which states that with probability $1 - \delta$, the true mean of the variable is at least within $\pm\epsilon$ of the observed variable average. The authors define ϵ as:

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

where r is a real-valued random variable with a range $R = r_{max} - r_{min}$ (i.e., the subtraction of the maximum and minimum values of r) considering the n independent observations of r.

One of the shortcomings of the HT derives from the fact that a single tree cannot accurately model complex learning problems. Even so, (FREUND; SCHAPIRE, 1997) presented a way to obtain a strong learner capable of modeling complex learning problems with an (expected) higher predictive performance without resorting to complex and intricate models. The proposed solution combines several weak learners strategically to make them function as one big learner (i.e., an ensemble). Even the weakest learners, who perform slightly better than random guessing, can form an ensemble. The good predictive performance has lead to an increasing interest in deploying ensemble models in real world problems involving data stream classification (GOMES et al., 2017a).

2.2 Ensembles for data stream classification

An ensemble combines several weak learners to improve the predictive performance of the model. When designing an ensemble, one of the main characteristics sought by researchers is the diversity of the ensemble members, particularly regarding the misclassifications, which means that every member should be as unique as possible (POLIKAR, 2006). When the models of an ensemble are unique, the ensemble is considered diverse, and its members complement each other. In essence, most models will provide correct predictions that compensate for the wrong predictions of a minority of models (GOMES et al., 2017a).

Many works in the literature have used ensembles in data stream learning problems with different goals, like concept-evolution (e.g., the apparition of novel classes) (MASUD et al., 2010; MASUD et al., 2011; FARID et al., 2013), recurring classes (AL-KHATEEB et al., 2012), and anomaly detection (TAN; TING; LIU, 2011), to cite a few.

A popular strategy to create ensembles is Bagging (BREIMAN, 1996). Although Bagging and its variants (e.g., Random Forest) are more than 20 years old, they are popular today. Bagging effectively reduces error without resorting to intricate models that are not trivial to train and fine-tune, such as deep neural networks. In contrast to Boosting (FREUND; SCHAPIRE et al., 1996), Bagging does not create dependency among the base models, facilitating the parallelization of the method in an online fashion. Besides that, Bagging variants yield higher predictive performance in the streaming setting than Boosting or other ensemble methods that impose dependencies among its base models. This phenomenon is present in several empirical experiments (OZA; RUSSELL, 2001; BIFET et al., 2009; BIFET; HOLMES; PFAHRINGER, 2010; GOMES et al., 2017b), and can be attributed to the difficulty of effectively modeling the dependencies in a streaming scenario as noted in (GOMES et al., 2017a).

2.2.1 Bagging ensembles for stream processing

Figure 1 shows an example of a bagging ensemble operating with a data stream. As the data arrives, the ensemble (big black rectangle) replicates the example and passes it to each (weak) learner, represented by the squares with gears inside on the left. Each learner processes the instance and outputs a prediction, portrayed by the colored circles. Then, the ensemble aggregates the predictions from every member using a heuristic and outputs the final prediction.

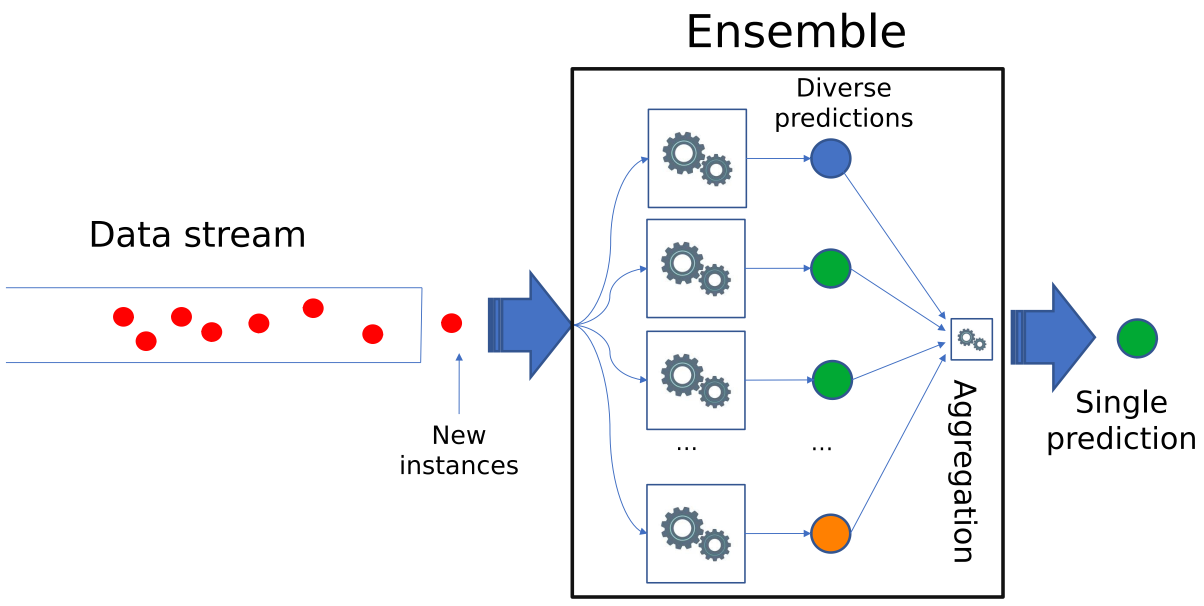


Figure 1 – Example of a Bagging Ensemble organization using majority vote.

Despite other decision tree algorithms (MANAPRAGADA; WEBB; SALEHI, 2018), the HT algorithm is often chosen as the base model for the online bagging algorithms. Even though the HT may not present the best predictive performance individually, it does yield reasonable accuracy without requiring excessive computational resources.

Next, we present a summary description of the original data stream adaptation for a Bagging algorithm (OzaBag) proposed by Oza and Russell (OZA; RUSSELL, 2001). In addition, we also present five ensemble algorithms inspired by OzaBag's idea.

Online Bagging (OzaBag - OB) (OZA; RUSSELL, 2001) is an incremental adaptation of the original Bagging algorithm. The authors demonstrate how to adapt the process of bootstrapping to a data stream setting. Online Bagging uses a Poisson($\lambda = 1$) distribution to assign weights to each incoming instance, which simulates the sampling with replacement from the original training set used on the standard bootstrapping. The weight received by the instance simulates the number of times it will be 'repeated' to simulate the bootstrapping process. A Poisson distribution with $\lambda = 1$ has a minimal range of possible values. One issue that may arise is the number of times the instances will receive weight zero (0), which means the learners will skip them during training. The chance of this happening is about 37%, which helps to approximate the offline version of bagging. In an online setting, this may be detrimental to performance (GOMES et al., 2017a). Therefore, other works (BIFET; HOLMES; PFAHRINGER, 2010; GOMES et al., 2017b) increase the number of times an instance is used for training by increasing the λ parameter.

OzaBag Adaptive Size Hoeffding Tree (OBagASHT) (BIFET et al., 2009) combines the OzaBag with Adaptive-Size Hoeffding Trees (ASHT). In an attempt to improve the diversity of the ensemble, – and, consequently, the predictive performance – the authors sought to enforce the creation of trees with different sizes. The algorithm receives a parameter indicating the tree's maximum number of nodes and applies some policies to prevent the tree from growing bigger than this parameter by deleting some nodes. The effect is the co-existence of different reset-speed trees in the ensemble. Smaller trees can adapt faster to the changes in data distribution, while larger trees can provide better predictive performance on data with little to no changes in distribution. Unfortunately, in practice, this algorithm did not outperform variants that relied on other mechanisms for adapting to changes, such as resetting learners periodically or reactively (GOMES et al., 2017a).

Online Bagging ADWIN (OBADWIN) (BIFET et al., 2009) combines OzaBag with the ADAptive WINdow (ADWIN) (BIFET; GAVALDà, 2007) change detection algorithm. When this algorithm detects a change in the data distribution, the ensemble replaces the learner with the lowest predictive performance with a new learner. ADWIN keeps a variable-length window of recently seen items. The property that the window has the maximal length is statistically consistent with the hypothesis that there has been no change in the average value inside the window. Such property implies that the average over the existing window can be reliably taken as an estimation of the current average in the stream at any time, except for a very small or very recent change that is still not statistically visible.

Leveraging Bagging (LBag) (BIFET; HOLMES; PFAHRINGER, 2010) extends OBADWIN by increasing the λ parameter of the Poisson distribution to 6, effectively causing each instance to have a higher weight and be used for training more often. While

OBADWIN has only one ADWIN detector for the whole ensemble, LBag maintains one ADWIN detector per model, allowing LBag to reset the models independently. This approach leverages the predictive performance of OBADWIN by using a higher weight, accelerating the training, and resetting the models individually. The faster training comes at the cost of bigger models that require more memory and processing time when compared to OB and OBADWIN. LBag also has a higher processing cost due to having one ADWIN detector for each model. In (BIFET; HOLMES; PFAHRINGER, 2010), the authors also attempted to further increase the diversity of LBag by randomizing the ensemble's output via random output codes. However, this approach was not very successful compared to maintaining a deterministic combination of the models' outputs.

Adaptive Random Forest (ARF) is an adaptation of the original Random Forest algorithm (BREIMAN, 2001) to the data stream setting. Random Forests (RF) use a technique called random subsets to increase diversity among the base models further. Each DT uses a different subset of attributes, allowing faster splits in leaf nodes and creating more unique learners. ARF uses HTs and simulates resampling the as in LBag (i.e., Poisson($\lambda = 6$)). The Adaptive part of ARF stems from the change detection and recovery strategies based on detecting warnings and drifts per tree in the ensemble. After signaling a warning, the ensemble creates and trains another model (a 'background tree') without affecting the ensemble predictions. If the warning escalates to a drift signal, the ensemble replaces the associated tree with its background tree. Notice that in the worst case, the number of tree models in ARF can be at most double the total number of trees due to the background trees. However, as noted in (GOMES et al., 2017b) the co-existence of a tree and its background tree is often short-lived.

Streaming Random Patches (SRP) (GOMES; READ; BIFET, 2019) is an ensemble method specially adapted to data stream classification, which combines random subspaces and online bagging. SRP is not constrained to a specific base learner as ARF since its diversity-inducing mechanisms are not built-in the base learner. In other words, SRP uses global randomization while ARF uses local randomization. The experiments focusing on Hoeffding trees showed that SRP could produce deeper trees, bringing the trade-off between increased diversity in the ensemble and computational cost.

2.3 Final considerations

This chapter presented an overview of data stream mining, its main characteristics and demonstrated how a decision tree algorithm can be adapted to work with data streams. The Chapter also introduced bagging ensembles by describing the main characteristics of six algorithms designed to simulate sampling with reposition (i.e., resampling) in a data stream setting. The next Chapter will provide an overview on parallelism and important aspects to improve the efficiency of algorithms.

Chapter 3

Parallel processing

A parallel computer is a set of processors, of varying scale, capable of working cooperatively to solve important computational problems. Once seen as an exotic subarea of computing, advances in computer science as a whole have elevated parallel processing status to a central aspect of the programming enterprise (FOSTER, 1995).

This phenomenon is not, by any means, a surprise, as researchers have been discussing models and algorithms for a long time (FLYNN, 1972; KARP; RAMACHANDRAN, 1989; DUNCAN, 1990).

Back in 1972, Flynn has proposed what would be known as the Flynn taxonomy. In that paper, Flynn examines the physical and logical attributes of computers to provide a simple classification based on the number of instructions and data that can be computed in parallel, as shown in Table 1. Later on, Duncan extended Flynn's taxonomy to incorporate more modern computer architectures.

	Single **I**nstruction	**M**ultiple **I**nstruction
Single **D**ata	SISD	MISD
Multiple **D**ata	SIMD	MIMD

Table 1 – Classification of parallel architectures proposed by (FLYNN, 1972).

It is a common practice to classify the solutions with regards to these architectures and in the case of this thesis it is straightforward to classify the solution regarding the instructions. We can confidently say that the algorithms proposed in this thesis use Multiple Instructions (MI). The explanation for this claim derives from the fact that the classifiers' structures differ among themselves. Such difference happens because each classifier uses a different subset of the data to perform the training and predicting phases. Using different data to train the models results in different structures to both, represent and process, the models, ultimately leading to different instructions for each model.

On the other hand, the classification of this thesis' work with regards to the data flows is more delicate, since the algorithms present both behaviors (single and multiple data flows) in different parts of the code. When data is arriving through the stream, it is not modified before replicating it to every classifier. Since each classifier consumes the same input through the data stream it constitutes in a single data behavior. Contrarily, the outputs produced by the classifiers before the aggregation step are different among themselves, which constitutes in a multiple data behavior.

One alternative to classify such problem is to chose the wider case, which leaves us with a MIMD problem with shared-memory as defined in Duncan's taxonomy. In fact, Duncan defines the MIMD architecture as multiple processors that can execute independent instruction streams, using local data. He highlights the potential of exploiting the independent nature of each processor and the cost-effectiveness compared to single-processor systems. On the other hand, other problems, such as data access synchronization and cache coherency, must be solved (DUNCAN, 1990).

The other alternative is to adopt the classification that is predominant in the algorithm behavior. In this case, the solution proposed in this thesis would be classified as MISD. During the training step, the only behavior present is the SD. In fact, the MD behavior is presented only in a brief interval when aggregating the predictions from the classifiers, which happens at the tail end of the classification step.

3.1 Designing parallel algorithms

Foster, defines four fundamental requirements for parallel software: concurrency, scalability, locality and modularity. Concurrency is defined as the ability of algorithms and program structures to leverage multiple processors and perform many operations at once. Scalability is related to having the resilience to increase the processor counts of a software. The third fundamental requirement, locality, is the ratio between local and remote accesses, where local memory accesses should be prioritized. Finally, modularity allows an easier management of processes and instructions (FOSTER, 1995).

Many models have been proposed to represent the parallel computation, task/channel, message passing, data parallelism, and shared memory. Each abstraction has its characteristics, but most of them use some form of task abstraction.

A common design methodology for parallel algorithms was defined by Foster. This methodology uses four stages: partitioning, communication, agglomeration, and mapping. Partitioning is the decomposition of the problem into small tasks with the goal of recognizing opportunities for parallel execution independently of physical characteristics. The communication stage defines the data dependency among the tasks created in the previious stage, as well as all the structures and policies required for them to operate. In the agglomeration stage, the task and communication structures are evaluated with respect to performance requirements and implementation costs. Multiple tasks can be combined into a larger task in this stage. Finally, the mapping stage assigns each task to a processor trying to extract the best performance possible. Mapping can be specified statically or determined at runtime by load-balancing algorithms.

By applying a methodology one can identify common problems in parallel algorithms, like granularity of the tasks, redundant computation, workload heterogeneity, scalability of task number and problem size, unbalanced communication, communication concurrency, among others. It should be noted that some problems may be either created or solved in the agglomeration and mapping stages (FOSTER, 1995).

Hennesy and Patterson highlight two challenges of parallel processing: the limited parallelism available in programs, in the sense that the programs have a limited parallel portion, and the relatively high cost of communications involved in remote access (HENNESSY; PATTERSON, 2012).

3.1.1 Task *vs* Data parallelism

According to Flynn's taxonomy (FLYNN, 1972), data-parallelism is considered a SIMD example, while task-parallelism falls into the MIMD category.

Data-parallelism exploits concurrency by applying the same operation to multiple elements of a data structure. Due to each operation on each data element being considered an independent task, data-parallelism possesses a naturally small granularity where the concept of locality does not apply. The downside is that programmers usually are required to provide explicit data distribution operations (FOSTER, 1995). Also, SIMD is potentially more energy efficient than MIMD (HENNESSY; PATTERSON, 2012).

Task-parallelism is more suited for heterogeneous computing, where different data structures must be processed with different instructions. Task-parallelism is more straightforward to develop, however their potential for improvement in performance is usually smaller than data-parallel alternatives. In the context of the present thesis only the task-parallel approach is used.

3.1.2 Shared *vs* Distributed memory

Another common distinction in parallel programs is related to the memory access. Even when working only with task-parallel solutions, there is a possibility that they will use shared or distributed memory.

A distributed-memory MIMD is a model where each processor has direct access to its own local memory only. The processors are interconnected by communication links, and exchange data by passing messages through those links. Examples of this model are

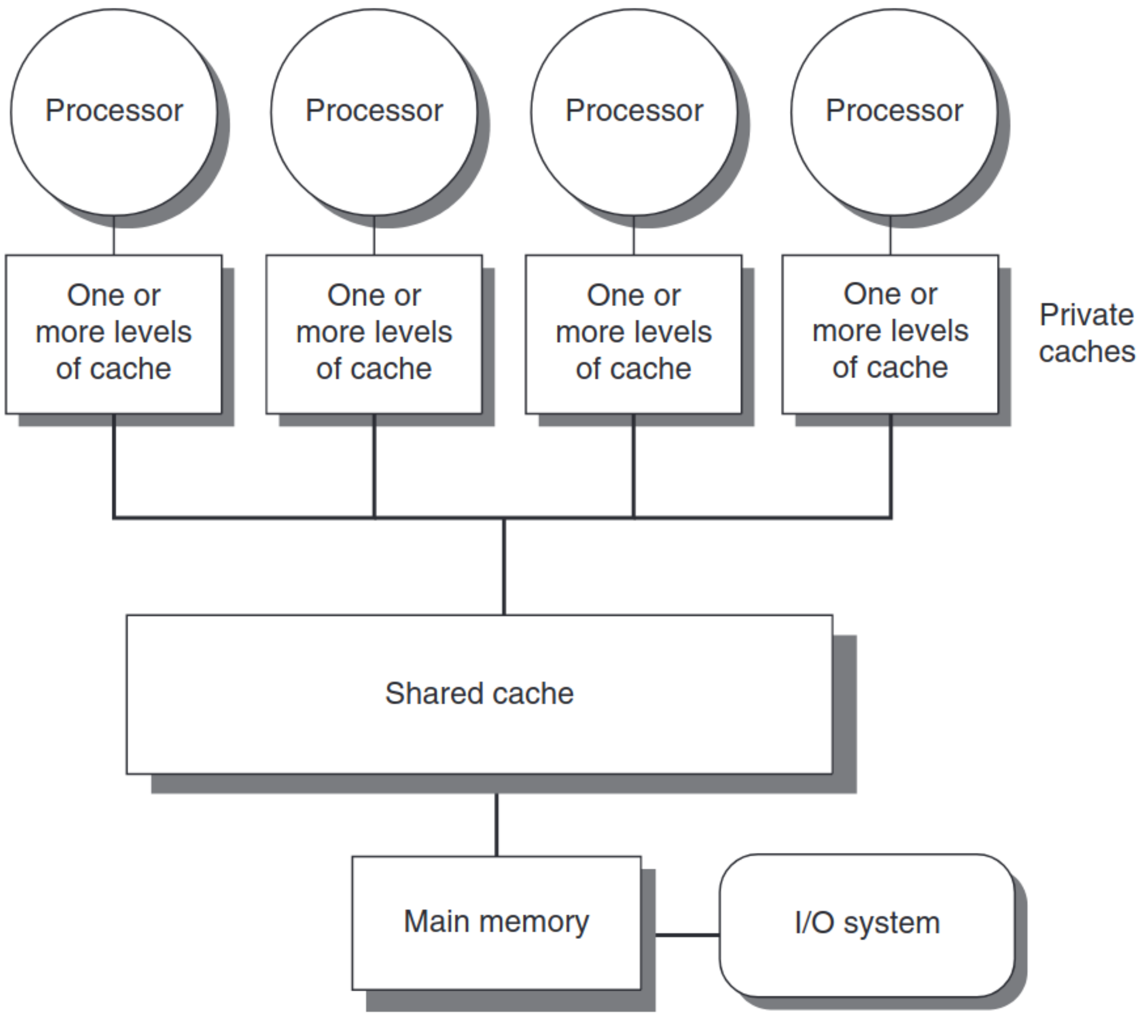

Processor
Processor
Processor
Processor
One or more levels of cache
One or more levels of cache
One or more levels of cache
One or more levels of cache
Private caches
Shared cache
Main memory
I/O system

Figure 2 – SMP multiprocessor (HENNESSY; PATTERSON, 2012).

clusters and grids, and has some characteristics like high costs and reliability through redundancy (HENNESSY; PATTERSON, 2012).

In the shared-memory programming model (SM-MIMD), tasks share a common address space, which they read and write asynchronously, while employing mechanisms such as locks and semaphores to control access to the shared memory. Unfortunately, challenges like understanding and managing locality become more difficult on most shared-memory architectures (FOSTER, 1995). Shared memory multiprocessors fall into two classes which are dependent on the number of processors and influence the memory organization and interconnect strategy.

(HENNESSY; PATTERSON, 2012) define two classes of multiprocessors: the symmetric multiprocessors (SMP) and distributed shared-memory multiprocessors (DSM). SMPs are also called centralized multiprocessors or uniform memory access (UMA) multiprocessors. They have a single memory unit that operates with a uniform latency for all processors. In SMPs any centralized resource in the system can become a bottleneck, with the usual shared L3 cache unlikely to scale much past eight cores. DSMs are also called

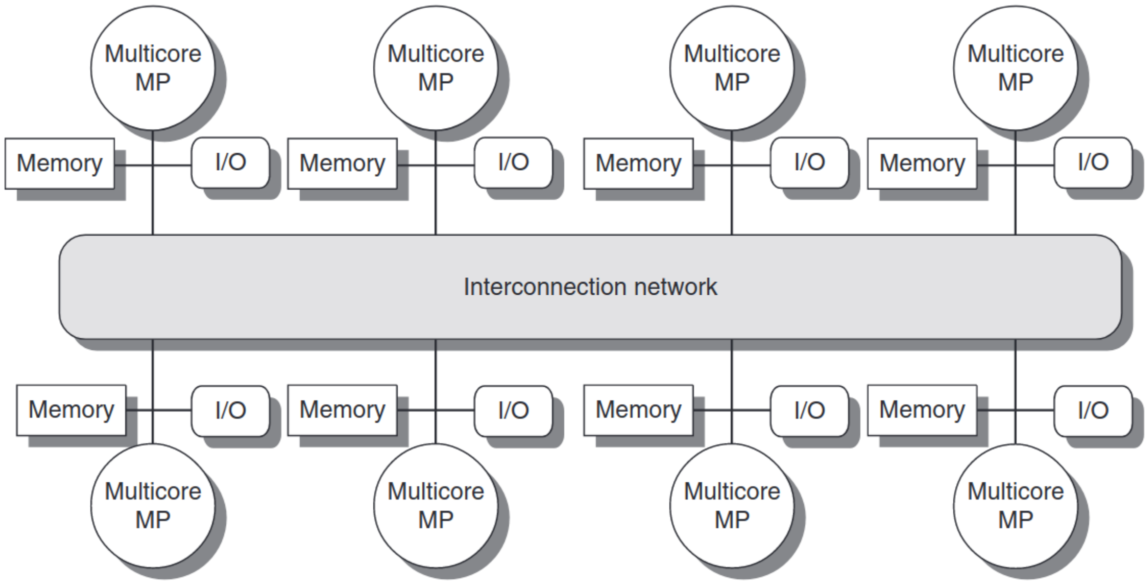


Figure 3 – DSM multiprocessor (HENNESSY; PATTERSON, 2012).

nonuniform memory access (NUMA) multiprocessors. They have a distributed memory in order to support the bandwidth demands from more processors. The downside is that by distributing the memory, it has a higher latency in general. Figures 2 and 3 shows examples of SMP and DSM multiprocessors, respectively.

3.2 Frameworks and tools

Originally created in 1954 by IBM for scientific and engineering applications, and later extended multiple times originating many different versions, the FORTRAN language has been a staple in the programming world (BACKUS, 1978). Given its target applications, it was expected that FORTRAN would be one of the pioneers in the High Performance area, which has lead to the creation of version like High Performance Fortran (HPF) (KENNEDY; KOELBEL; ZIMA, 2007) – a representative of the data-parallel programming model – and Fortran M (FM) – a representative of the task-parallel programming model (FOSTER, 1995).

Other staples of the HPC field are the C and C++ languages. C was created with the ability to work with task-parallelism through the famous POSIX threads (BUTENHOF, 1997), which provide minimal functionality and for this reason some consider this approach the assembly language of parallelism.

Two famous and widely used APIs capable of providing tools to develop parallel programs are the MPI (FORUM, 1994) and OpenMP (CHANDRA et al., 2001). MPI is the standard on communication over message passing. It is usually deployed on distributed memory, however it is possible to use it implements a shared-memory model. OpenMP is mainly based on simple compiler directives used to guide mostly the parallelization of regular loops. Recent expansions provide more mechanisms to improve the control over the application execution.

Although it was never seen as a high performance language, Java offers tools that allow the programmer to implement task-parallelism with relatively ease through resources available on the concurrent package. It offers mechanisms to implement and manage Threads, going from very simple models, to more complex behaviors.

The Java parallel APIs are based on the Fork-Join abstraction model, which allows the expression of parallel implementations without prior knowledge on the target system. This model is composed by three steps: fork, computation and join. First, in the fork step new threads are created on demand. Then, in the computation step, each thread executes one or more tasks. Finally, in the join step the parallel threads synchronize and finish before continuing the sequential region of the program. This fork-compute-join process can be repeated many times during the execution of a program. Aiming to reduce the overhead of thread creation/destruction, Fork-Join implementations usually employ thread pools which support forked tasks management. These pooled threads are not destroyed when the task finishes but instead release resources and become idle (EKANAYAKE et al., 2016).

There are two frameworks that implement the Fork-Join thread model in Java. The framework **Executor Service (ES)** is available since Java 7. It has methods to track the progress of a task and manage the task's termination. The main goal of this framework is to facilitate thread management through the creation of a Service with a fixed thread pool size, reserving and reusing these threads. Once a service has been created, tasks can be invoked by passing Runnables/Callables for it.

The second framework is the **Java Fork/Join (FJ)**, implemented on the ForkJoinPool[1] class, also available since Java 7. It is built upon the ExecutorService, and supports additional abstractions. Among the additions, a built-in work-stealing algorithm, capable of supporting load-balancing, allows idle threads to steal work from the queues of other threads in the pool. The Fork/Join framework was designed with problems split in recursive tasks as its target applications. This conceptual decision is, perhaps, the biggest difference between both frameworks.

In this thesis, we have used, initially, both frameworks. However, as the work progressed, ExecutorService was deemed a better option.

3.3 Data locality

Many authors consider data locality (also referenced as Memory Locality, sometimes) a fundamental property and design principle for optimizing the performance of hardware, software, and algorithms (YUAN et al., 2019; FOSTER, 1995). Locality can be defined

1 <https://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html>

as “the tendency for programs to cluster references to subsets of address space for extended periods” (DENNING; MARTELL, 2015). Due to the increasing gap in processor and memory speeds (JACOB; WANG; NG, 2010), locality has played a central role in optimizing the performance of operating systems over the decades.

Several metrics have been proposed to measure and quantify the locality of data (YUAN et al., 2019). The notion of *reuse distance* (RD) is based on direct and unambiguous measurements, do not depend on idealistic assumptions, and are extensions of observational stochastic (YUAN et al., 2019). Thus, RD can be used to evaluate how mini-batching improves the performance and resource (i.e., cache) sharing of bagging ensembles implementations.

From a historical perspective, memory locality has been studied over decades to optimize the memory hierarchy, operating systems, software, and algorithms design (DENNING, 1968; DENNING; SLUTZ, 1978; SLUTZ; TRAIGER, 1974) with recent advances in measurement techniques (IBRAHIM; STROHMAIER, 2010), trace generation (SHEN; SHAW, 2008), and formal modeling (BALAKRISHNAN; SOLIHIN, 2012; MAEDA et al., 2017). In a recent work (YUAN et al., 2019), Yuan et al. built upon previous works by proposing the *relational theory of locality* (RTL), a theoretical framework that unifies several memory locality measures used along five decades of study and research in the field. RTL provides mathematical background and categorizes the measures in three different types of locality. The authors showed how such measures relate to each other and whether and how they can be inter-converted.

Next, we discuss memory locality of a stream processing system operating according to the algorithms described in the previous section. Each algorithm implements an ensemble L, composed by a set of learners $l_i \in E$. We refer to an individual learner as l_i $(1 \leq i \leq |E|)$. A *stream* S is a countably infinite set of data elements $s \in S$. Each stream element s:$\langle$v, t$\rangle$ consists of a relational tuple v conforming to some schema, with an application time value $t_i \in T$. We assume that the time domain T is a discrete, linearly ordered, countably infinite set of time instants $t \in T$. As the stream is potentially infinite, we assume that T is bounded in the past, but not necessarily in the future. Thus, due to memory limitations and response time constraints, the algorithms need to incrementally process incoming data elements in a single pass, performing both classification and training as data elements arrive.

A *trace* N is a sequence of references to data or memory locations denoted by $N = mt(1, \ldots, n)$, where n is the trace’s length. A trace can access a set of m distinct memory addresses, while the set of distinct memory addresses is be denoted by $M = e_1, \ldots, e_m$, where m is the number of distinct memory addresses in M. The model allows abstracting from any granularity issues so that a data item may be either a variable, a data block, a page, or an object. For illustration, we can use some trace examples composed of just three data elements a, b, c, including those repeating them once in the same order (i.e., *abc abc*), in the opposite order (i.e., *abc cba*), or repeating them indefinitely (i.e., *abc abc* . . .).

3.3.1 Improving access locality

In essence, access locality is related to measuring the locality for each memory access. From the five definitions of access locality provided by YUAN et al. (YUAN et al., 2019), we use only the definition of *reuse distance sequence*, or *reuse distance* (RD) for short, because it suffices to demonstrate that mini-batching can improve ensemble implementations' access locality. The equivalence among the definitions is proven in (YUAN et al., 2019).

The reuse distance (RD) is defined as "the number of distinct data accessed since the last access to the same datum, including the reused datum" (YUAN et al., 2019). The reuse distance is ∞ for its first access. For a finite reuse distance, the minimum is 1 (because it includes the reused datum), and the maximum is m. For example, the RD sequence is $\infty\infty\infty$ 333 for *abc abc* and $\infty\infty\infty$ 135 for *abc cba*.

Before demonstrating the benefits of mini-batching, it is worth noting that stream processing ensembles have two principal operations: the classification (in line 5 of Algorithm 1) and the training (line 12 of Algorithm 1). The former reads a few model variables of each learner, while the latter is (by far) the dominant operation in terms of computational cost because it performs both read and write operations to update the learner's models. For this reason, our analysis presented here focus on the training operation.

For the sake of illustration, Table 2 presents a simple example with an ensemble of $m = 4$ learners processing a stream of $n = 6$ data items. Without mini-batching, the processing of the first data item produces a sequence of m occurrences of ∞ reuse distance. However, for finite reuse distance, the minimum reuse distance is 1 because it includes the reused datum, and the maximum is m. Thus, ∞ is shown only for illustration, being ignored in our analysis hereafter. In this example, for each access e_i the reuse distance is equal to the number of ensembles m. With mini-batching, each ensemble is accessed exactly once within the mini-batch (with reuse distance m) and reused $b - 1$ times. One could easily realize the benefits of mini-batching by simply substituting every ∞ by 1 and calculating the average reuse distance for the two executions. Next, we demonstrate the benefits.

For the proofs shown in this section we assume the amount of memory used to implement the ensemble exceeds the cache memory size (which is quite realistic). Otherwise, all accesses will hit the cache and the order in which memory positions are accessed does not influence cache misses.

Theorem 1. *The reuse distance of an ensemble of m learners processing a n-length data stream is $\mathcal{O}(nm^2)$.*

Table 2 – Example: A stream of $n = 6$ data items being processed by an ensemble of m learners without and with mini-batching.

RD without mini-batching. A semicolon (;) denotes separation between data items.

Access sequence	$e_1, e_2, \cdots, e_m; e_1, e_2, \cdots, e_m; e_1, e_2, \cdots, e_m; e_1, e_2, \cdots, e_m; e_1, e_2, \cdots, e_m; e_1, e_2, \cdots, e_m$
RD sequence	$m, m, \cdots, m; m, m, \cdots, m; m, m, \cdots, m; m, m, \cdots, m; m, m, \cdots, m; m, m, \cdots, m$

RD with mini-batching of size $b = 3$. A semicolon (;) separates mini-batches.

Access sequence	$e_1, e_1, e_1; e_2, e_2, e_2; \cdots; e_m, e_m, e_m; e_1, e_1, e_1; e_2, e_2, e_2; \cdots; e_m, e_m, e_m$
RD sequence	m, 1, 1; m, 1, 1; ···; m, 1, 1; m, 1, 1; m, 1, 1; ···; m, 1, 1

Proof. Consider an ensemble composed of m learners and a data stream composed of n data elements. Let $e_1, e_2, \ldots, e_m$ be the memory locations accessed during a sequential execution of the ensemble to process. As the model can express arbitrary granularity, for simplicity consider that e_i denotes an access to the data structures of the *i-th* learner l_i of the ensemble.

The execution of the training operation (line 5 of Algorithm 1) will produce the access sequence $(e_1, e_2, \ldots, e_m)$ for each arriving data instance because it invokes all the learners' training in this exact order. Thus, the training operation will produce the access sequence $(e_1, e_2, \ldots, e_m)^n$ for the access stream. Then, considering finite distances (i.e., substituting ∞ by m), the reuse distance sequence will be $(m)^m$ for all the arriving data items. Then, we can sum up the entire sequence to obtain RD as follows:

$$RD = \sum_{1}^{n} \sum_{1}^{m} m = nmm = \mathcal{O}(nm^2). \tag{2}$$

□

Next, we can estimate the benefit of mini-batching (as described in Algorithm 3) for reducing the reuse distance.

Theorem 2. *Mini-batching can reduce the reuse distance of an ensemble implementation by a constant factor.*

Proof. Consider an ensemble implementation like Algorithm 3, whose computational cost is dominated by the training phase. With mini-batching, the access sequence will change from $(e_1, e_2, \ldots, e_m)^n$ (in Algorithm 1) to $({e_1}^b, {e_2}^b, \ldots, {e_m}^b)^{n/b}$ (in Algorithm 3), where b is the mini-batch size. For each mini-batch, the RD sequence is $(m, 1^{b-1})^m$. Finally, the RD sequence for the whole stream will be $((m, 1^{b-1})^m)^{n/b}$, and the RD can be computed as:

$$RD = \sum_{1}^{n/b} \sum_{1}^{m} (m + b - 1) = \mathcal{O}(\frac{nm^2}{b}). \tag{3}$$

Hence, the mini-batch can reduce the reuse distance by a constant factor of b, where b is the mini-batch size. □

Although our demonstration assumes sequential processing, the result is also valid for the parallel execution proposed in Algorithm 3. Notice that the outer loop in line 11 assigns a different ensemble learner for each processing core, while the innermost loop iterates over the mini-batch data items. Thus, each processing core needs to load only one learner model in its memory caches to process the entire mini-batch.

Table 3 – Parallel execution of a stream of $n = 6$ data items being processed by an ensemble of $m = 4$ learners in 3 processors with mini-batch size $b = 3$.

	Reuse distance with mini-batching of size $b = 3$	
P1	Access seq.	e_1,e_1,e_1,e_1,e_1,e_1 e_4,e_4,e_4,e_4,e_4,e_4,. . .
	RD seq.	4, 1, 1, 1, 1, 1, 4, 1, 1, 1, 1, 1, . . .
P2	Access seq.	e_2,e_2,e_2,e_2,e_2,e_2,. . .
	RD seq.	4, 1, 1, 1, 1, 1, . . .
P3	Access seq.	e_3,e_3,e_3,e_3,e_3,e_3,. . .
	RD seq.	m,1, 1, 1, 1, 1, . . .

It is worth noting such results hold regardless of the locality measure used for the demonstration. As formally demonstrated by YUAN et al. (YUAN et al., 2019), locality access measures such as *address independent (AI) sequence*, *reuse interval (RI) sequence*, *per datum sequence of reuse interval (PD-RI)*, and *per datum reuse distance (PD-RD)* are equivalent to each other, and they can be inter-converted. Using these results, other measures could be seamlessly used to demonstrate that mini-batching improves access locality of the implementation of ensembles for stream processing. We chose the measure *reuse distance* because of its close relation to cache misses, as the larger the reuse distance, the higher the cache misses. Cache misses occur when the reuse distance is big enough to fill the cache memory.

Theorem 3. *Mini-batching provides optimal access locality for the implementation of ensembles.*

Proof. The proof is straightforward. With mini-batching, at least one mini-batch (of length n) is needed to contain all the stream elements. For each ensemble learner l_i, the processing of the first data element e_1 in the mini-batch will result in reuse distance of m, because the l_i's data structures are being touched for the very first time. For all the remaining $b-1$ data elements of the mini-batch the reuse distance will be 1 as it reuses the same datum. Thus, every learner produces a reuse distance of $m+b-1$. This is a lower bound on the reuse distance as no other order can reduce it. For an ensemble of m learners, the total reuse distance will be $m*(m+b-1) = \mathcal{O}(m^2)$ as b is a constant. This is equal to Eq. 3 when the batch size b is equal to the stream length n.

□

Although using only one single mini-batch to process the entire data stream is useful for demonstrating that the access locality is optimal, it is not useful in practice. Notice that the pure stream processing (as in Algorithm 1) performs both the classification and training steps for every stream's incoming data item. Thus, the learner models continuously evolve, and the processing of every data instance can influence the next incoming data classification. On the other hand, with mini-batching (as proposed in Algorithm 3), the ensemble training is deferred to the end of each mini-batch. So, setting a mini-batch size of 1 boils down to pure stream processing. In contrast, mini-batches of the same length as the entire data stream turns it into a pure batching scheme in which all data instances are classified using models built during an offline training phase that precedes the entire stream. In summary, the choice of the mini-batch size raises a trade-off between learning capabilities (with short mini-batches) and computational performance (with larger mini-batches).

Yuan et al. (YUAN et al., 2019) demonstrated that several locality measures are equivalent and may be inter-converted. However, not all measures can be equally usable for our purpose. We chose the *reuse distance* because the demonstration that mini-batching leads to optimal locality becomes straightforward with this measure, but not using the footprint or miss ratio, for instance.

3.4 Final considerations

This chapter presented an overview of the basic taxonomy of parallel architectures. Understanding the main characteristics of the architectures and how the components work together is important to design an efficient algorithm. Also, this chapter presented an overview of data locality, an important concept regarding the efficiency of algorithms. In the next Chapter, we explore the related work of the thesis divided into three subareas: parallel ML, mini-batching and energy efficiency.

Chapter 4

Related work

Research on the parallelization of machine learning methods dates to the early 90s (CHAN; STOLFO, 1993), when most papers focused on batch ML methods that require the whole dataset in the main memory to train the model. In early data mining, the batch learning approach is applied by processing the whole training data (one or multiple times) to output the decision models. Then, the decision models can be applied to new production data. This is usually referred to by batch learning (GAMA, 2012), batch-mode algorithms (SILVA et al., 2013), static data mining (KRAWCZYK et al., 2017b), among others in the literature. From now on we will use batch learning to refer to non-stream learning methods.

Over the years, with the increase in computational power, the focus shifted from single classifiers (SCHöLKOPF; PLATT; HOFMANN, 2007; WANG, 2016; JOSHI; KARYPIS; KUMAR, 1998) to ensembles. In the context of ensemble learners, many studies have been made on MapReduce (MR) frameworks (YAN et al., 2009; BASILICO et al., 2011; PANDA et al., 2009; JAHNKE, 2009; XAVIER; THIRUNAVUKARASU, 2017; LIU, 2014; GHOTING et al., 2011), which are not suitable for applications with requirements of low response times, even being capable of processing huge amounts of data with high scalability (SENGER et al., 2016). Another investigation approach is the use of GPUs to process ensembles (LIAO et al., 2013; SAFFARI et al., 2009), however GPUs are better for data-parallel problems.

Among the works that explored multi-core parallelism, distributed or not, we can further subdivide it in batch (HAJEWSKI; OLIVEIRA, 2020; ISLAM et al., 2009; CYGANEK; SOCHA, 2014; VALLE et al., 2010; HOYOS-IDROBO et al., 2018; NOJIMA; MIHARA; ISHIBUCHI, 2010; WEILL et al., 2019; HUSSAIN et al., 2012; JIN; AGRAWAL, 2003a) or data stream (HORAK; BERKA; VAJTERSIC, 2013; MARTINOVIC et al., 2019; QIAN et al., 2016; MARRóN et al., 2017) methods. The Message Passing Interface (MPI) standard was used in many works with various ensemble methods, such as: ensemble of improved and faster Support Vector Machine (SVM) (HAJEWSKI; OLIVEIRA, 2020), ensemble of Neural Networks (NN) (VALLE et al., 2010), ensemble of fuzzy rule generation (NOJIMA; MIHARA; ISHIBUCHI, 2010), bagging decision rule ensembles (HORAK; BERKA; VAJTERSIC, 2013) and regression ensembles (MARTINOVIC et al., 2019). A miscellaneous of tools and scopes can be found in the remaining literature. (ISLAM et al., 2009) proposed a multi-classifier ensemble consisting of three classifiers which are pinned to a thread. (CYGANEK; SOCHA, 2014) implemented multi-classifiers using OpenMP. An ensemble of SVMs was implemented using joblib (a Python multiprocessing lib) and scikit-learn by (HOYOS-IDROBO et al., 2018). (WEILL et al., 2019) used TensorFlow to build a scalable and extensible framework for ensembles parallelization. (JIN; AGRAWAL, 2003a) proposed an efficient Random Forest (RF) implementation that improves memory access due to better data representation on machines that combine both shared and distributed memory; it was implemented using FREERIDE (previous work from the authors). (QIAN et al., 2016) parallelizes an ensemble of J48 for grid platforms using Java. (MARRóN et al., 2017) implemented a low-latency Hoeffding Tree (HT) in C++ and used it in RFs. In general, the related works mentioned so far differ from this thesis in two main aspects: they either focus on batch approaches (i.e., they do not focus on stream processing) or focus on the implementation and performance aspects of ensembles of a specific type of classifier.

A study the impact of concurrency on memory access pattern and performance of ensembles is presented by (HORAK; BERKA; VAJTERSIC, 2013). The authors proposed a two-stage bagging architecture that combines single-class recognizers with two-class discriminators to improve accuracy and allow parallel processing. They also addressed load balancing for the parallel classifier construction and used the algorithm SCALLOP as the base for the experiments to validate the architecture implemented in MPI. (MARTINOVIC et al., 2019) enhanced a dynamic auto-tuning framework in a distributed fashion, by using two strategies. The authors used a scalable infrastructure capable of leveraging the parallelism of the underlying platform for ensemble models to speed up the predictive capabilities, while iteratively gathering production data. Results show that the approach implemented in MPI is able to learn the application knowledge by exploring a small fraction of the design space. A novel ensemble for data stream classification is proposed by (QIAN et al., 2016). This solution maps different raw data to multiple grids, where the first-order geometric center is used to represent and classify data. This method performs data compression which, in turn, increases the accuracy and computational efficiency. It was implemented in Java and tested in both multi-core and grid environments. (MARRóN et al., 2017) propose an implementation of RF-based on vector SIMD instructions and changes the representation of Binary Hoeffding Trees (HT) to fit into the L1 cache. It was implemented in C++ and benchmarked against MOA and StreamDM using two real and eleven synthetic datasets. It is noteworthy that the authors compare the performance of a single tree and the ensemble using different hardware architectures.

In reality, these four works are more closely related to this thesis as they approach the performance of ensembles in the context of data streams. However, they differ in the following aspects. The solutions presented by (HORAK; BERKA; VAJTERSIC, 2013) and (MARRóN et al., 2017) focus on specific algorithms, SCALLOP and Binary HT, respectively. The focus of he solution proposed by (QIAN et al., 2016) is focused on data compression, while (MARTINOVIC et al., 2019) leverages parallel processing to improve the parameters of the algorithm.

Table 4 summarizes the literature on parallelization of ML methods with focus on data streams and ensembles.

Reference	Tool	Method	Algorithm	Platform
(YAN et al., 2009)	MR	Batch	Subspace Bagging	Hadoop
(HAJEWSKI; OLIVEIRA, 2020)	MPI	Batch	Ensemble SmothSVM	Distributed
(BASILICO et al., 2011)	MR	Batch	Ensemble RF	Hadoop
(PANDA et al., 2009)	MR	Batch	RF	Hadoop
(ISLAM et al., 2009)	Pthreads	Batch	Multi-classifier (3) ensemble	Multi-core
(CYGANEK; SOCHA, 2014)	TensorFlow OpenMP	Batch	Multi-classifier (1-13) ensemble	Multi-core
(HORAK; BERKA; VAJTERSIC, 2013)	MPI	Stream	Bagging of SCALLOP	Multi-core
(VALLE et al., 2010)	MPI	Batch	Ensemble NN	Distributed
(HOYOS-IDROBO et al., 2018)	Sci-kit learn joblib	Batch	Ensemble SVM	Multi-core
(JAHNKE, 2009)	MR	Batch	Different ensembles	Hadoop
(NOJIMA; MIHARA; ISHIBUCHI, 2010)	MPI	Batch	Ensemble Fuzy	Distributed
(QIAN et al., 2016)	Java	Stream	Ensemble J48	Distributed
(LIAO et al., 2013)	PyCUDA Parakeet	Batch	RF	GPU
(XAVIER; THIRUNAVUKARASU, 2017)	-	Batch	RF	Spark
(SAFFARI et al., 2009)	-	Stream	RF	GPU
(WEILL et al., 2019)	TensorFlow	Batch	Ensemble TensorFlow	Distributed & GPU
(MARTINOVIC et al., 2019)	MPI	Stream	Ensembles Regression	Distributed
(LIU, 2014)	MR	Batch	Adaboost	Hadoop
(GHOTING et al., 2011)	MR	Batch	RF	Hadoop
(SCHöLKOPF; PLATT; HOFMANN, 2007)	MR	Batch	Single model	Multi-core
(HUSSAIN et al., 2012)	FPGA	Batch	Single model	FPGA, Multi-core
(WANG, 2016)	-	Batch	Ensemble KNN	Multi-core
(BEN-HAIM; TOM-TOV, 2010)	MPI	Stream	Single model	Distributed
(CHAN; STOLFO, 1993)	-	Batch	Meta-learners	Multi-core
(JIN; AGRAWAL, 2003a)	-	Batch	RF	Distributed
(JIN; AGRAWAL, 2003b)	-	Stream	Numerical pruning and single model	Sequential Stream
(JOSHI; KARYPIS; KUMAR, 1998)	MPI	Batch	Single model	Distributed
(MARRóN et al., 2017)	MPI CPP	Stream	RF of binary trees	Multi-core

Table 4 – Summary of related works in parallelized machine learning methods.

4.1 Mini-batching

Besides parallelization, there are other alternatives to improve the performance of machine learning applications. One such alternative is to optimize the performance of machine learning applications using some form of mini-batching. In summary, mini-batching consists of processing small chunks containing several data instances to be processed at once, instead of processing a single instance at a time. Next, we present related studies that have taken this approach.

Variations of mini-batching have been employed with different goals. For instance, stream processing systems such as Spark and Flink group data in small batches to improve performance and fault-tolerance (ZAHARIA et al., 2012; CARBONE et al., 2015). (WANG et al., 2012) proposed a scheduling strategy to find energy-optimal batching periods for real-time tasks with deadline constraints to execute on heterogeneous sensors. Comet, proposed by (HE et al., 2010), is a stream processing system that identifies the optimal sizes of batches of data items to be processed for large-scale data streams. The proposal is based on a model named *Batched Stream Processing* (BSP) that focuses on modeling recurring (batch) computations on incrementally bulk-appended data streams. Despite some similarities, this work focuses on the reuse of input data and intermediate results to reduce recomputing and I/O redundancies that cause bandwidth waste. Similar techniques that group work units into small batches have been used in other application areas, such as in web search engines (BONACIC et al., 2015; GAIOSO et al., 2019) or content delivery applications (JAYASUNDARA; GOPALAKRISHNAN, 2013). In summary, these works use mini-batching for grouping processing units into larger ones that increase the utilization of resources in multi-core or distributed processing systems.

Another mini-batching approach is widely used in gradient descent-based techniques for training deep learning and several machine learning methods based on optimization approaches (GOODFELLOW; BENGIO; COURVILLE, 2016; WITTEN; FRANK, 2002). Similar mini-batch approaches can be used in many inversion problems. For instance, (KUKREJA et al., 2020) use mini-batches to propose a new method that combines checkpointing with error-controlled lossy compression for large-scale high-performance inversion problems. The method reduces movement, allowing a reduction in run time as well as peak memory. In general, such methods use mini-batching to achieve a good trade-off between the amount of information used and computational costs for the optimization process used for training the learners. This is different from the approach proposed in this thesis, which focuses on the use of mini-batching for improving data locality.

(ZHANG et al., 2019) proposed two scheduling strategies to reduce both the delay and energy consumption of executing small batches of Deep Neural Networks (DNN) tasks on edge nodes such as IoT devices. Although the strategies can be applied to CPUs, the proposal focuses on executing DNN applications on GPU devices in the edge. Their focus is to optimize resource utilization in the edge of the network.

A method that optimizes ensembles of Artificial Neural Networks (ANNs) and whose computational and memory costs are significantly lower than typical solutions is proposed by (WEN; TRAN; BA, 2020). The improvement is achieved by defining each weight matrix to be the Hadamard product of a shared weight among all ensemble members and a rankone matrix per member. Their method yields competitive accuracy and uncertainties as typical ensembles, achieving 3X speedups at test time and 3X less memory for ensembles of 4 learners. This work focuses only on NN ensembles to classify image datasets.

Table 5 summarizes the literature on mini-batching approaches.

Reference	Objective	Environment
(ZAHARIA et al., 2012)	Fault tolerance and performance	Apache Spark
(CARBONE et al., 2015)	Fault tolerance and performance	Apache Flink
(WANG et al., 2012)	Energy optimization	Real-time tasks on heterogeneous sensors
(HE et al., 2010)	Reduce recomputing and IO redundancies	Large-scale data streams
(BONACIC et al., 2015)	Increase resource utilization	Web search engines
(GAIOSO et al., 2019)	Increase the utilization of resources	Web search engines
(JAYASUNDARA; GOPALAKRISHNAN, 2013)	Increase the utilization of resources	Content delivery applications
(GOODFELLOW; BENGIO; COURVILLE, 2016)	Improve data quality	Gradient descent based methods
(WITTEN; FRANK, 2002)	Improve data quality	Gradient descent based methods
(KUKREJA et al., 2020)	Reduce data movement	Large-scale FWI
(ZHANG et al., 2019)	Reduce delay and energy consumption	DNNs on the edge
(WEN; TRAN; BA, 2020)	Reduce data on weight matrix	ANNs for image classification

Table 5 – Summary of related works that used mini-batches

4.2 Energy efficiency

Energy efficiency is another field of research that is related to this thesis. (MARTIN; LAVESSON; GRAHN, 2015) emphasizes energy consumption and energy efficiency as important factors to consider during data mining algorithm analysis and evaluation. The work extended the CRISP (Cross Industry Standard Process for Data Mining) framework to include energy consumption analysis, demonstrating how energy consumption and accuracy are affected when varying the parameters of the Very Fast Decision Tree (VFDT) algorithm. The results indicate that energy consumption can be reduced by up to 92.5% while maintaining accuracy.

An analyzis of power consumption for both batch and online data stream learning was made by (AMEZZANE et al., 2019). The authors experimented with three online and three batch algorithms. Among their conclusions is the finding that the CPU consumes up to 87% of the total energy. Although evaluating online learners, this work tested only single model classifiers.

A comparison of the four-way relationship among time efficiency, energy consumption, predictive performance, and memory costs is presented by (LOPES et al., 2020). The comparison is made by tuning the hyper-parameters of VFDT, SVFDT, and SVFDT with OLBoost. The work demonstrated that the most complex method delivers the best predictive performance at the expense of worse memory and energy performance.

A modular, scalable, and efficient FPGA-based implementation of kNN for System on Chip devices is presented by (VIEIRA; DUARTE; NETO, 2019). The solution shows improvements of 60X in execution time and 50X in energy efficiency.

An energy-efficient approach to real-time prediction with high levels of accuracy called *nmin adaptation* is presented by (GARCíA-MARTíN; BIFET; LAVESSON, 2021). This reduces the energy-consumption of Hoeffding Trees ensembles by adapting the number of instances required to create a split. This method can reduce energy consumption by 21% on average with a small impact on accuracy. They also presented detailed theoretical energy models for ensembles of Hoeffding trees and a generic approach to creating energy models applicable to any class of algorithms.

4.3 How this thesis is different from the literature

We have brought three sub-areas of related work to compare with this thesis, parallelization of ensemble algorithms, usage of mini-batching to improve performance of ML, and energy efficient ML. Although several parallel ensemble algorithms have been proposed, methods focusing on their efficient implementation are seldom approached in the related literature. In particular, studies of memory access locality for improving the performance of ensembles are rarely approached. To date, this is the first work to propose a strategy for improving memory access locality for parallel implementation of bagging ensembles on multi-core systems. This thesis employs both measurement techniques and theoretical foundations proposed in (YUAN et al., 2019) to demonstrate the benefits of mini-batching for the implementation of ensembles.

Another difference of this thesis compared to previous work is the focus on a class of ensemble algorithms composed of bagging ensembles executing in the context of data streams. Furthermore, the strategy proposed in this thesis is orthogonal to any optimization and parallel implementation of a specific learning algorithm within the ensemble. Being orthogonal, the mini-batching approach for ensemble optimization can be combined with other parallelization/optimization strategies that focus on specific learner algorithms within the ensemble can be investigated, with potential benefits for each other.

Regarding energy efficiency, the literature that evaluates it is scarce and proposals tend to work with stand-alone models (instead of ensembles). In addition, few works evaluate data stream algorithms in a more realistic deployment where data is really transmitted through the network at parameterizable loads. In summary, this thesis measures the energy consumption of several bagging ensembles at different levels of load (throughput), with data being received through the network, to provide a better energy consumption profile for the online data stream learning context.

Chapter 5

Parallelization of bagging ensembles for data streams

Ensembles are a class of ML algorithms that achieve remarkable predictive performance at the expense of more computational resources. This characteristic makes ensembles more likely to violate constraints imposed in a data stream setting, such as response time. However, the natural organization of ensembles offer an opportunity for parallelism, as each member has to classify and update its model using the same data.

Although the members of the ensemble may be homogeneous in type, ensemble algorithms are not amendable for data parallelism. As stated before, diversity among ensemble members, particularly in regards to misclassification, is a desired trait. It is inevitable that a diverse ensemble will have members with different structures and organizations, making the data-parallel model hard to implement without further modifications and adjustments. On the other hand, task-parallelism can naturally be applied on ensembles where each member executes independently and does not rely on communication with the others. Bagging ensembles possess such characteristics, making them great candidates for implementing parallelism.

Therefore, we seek to improve the performance of bagging ensembles while adhering to the data stream setting constraints. Next, we present our assumptions to provide a better definition of the scope of this thesis, including, but not limited to, the evaluation of the proposal.

To define the scope in a single sentence, we need accurate ensemble methods, capable of working with data streams, and easy to evaluate. The reasons for implementing task-parallelism in ensembles are already known. The only requisite left to define is the evaluation framework. Since we want to implement prototypes of our solution in many different algorithms to test if our technique can be applied to a whole class of ensembles, we need several baseline sequential ensemble implementations. Instead of implementing the algorithms and evaluation from the ground, we have chosen to use the existing framework Massive Online Analysis (MOA)[1] because it allows the reuse of several bagging ensemble designed for data stream processing. By reusing MOA algorithms, we provide a seamless and reproducible evaluation of our proposal in a framework that has been used for many studies in the area (BIFET et al., 2010). Using MOA also allows us to focus on the efficiency of the implementation instead of the correctness of the baseline, as long as we keep the baseline predictions intact. The MOA framework is implemented in Java, which is not a language that focus on implementing either energy-efficient or high-performance applications. Even so, the work in (PEREIRA et al., 2017) shows that Java is in the top 5 languages (out of 27 tested) that need less energy and time to execute the applications.

5.1 Parallel implementation

We propose and implement a parallel algorithm for bagging ensembles in Algorithm 1. The objective of this implementation goes beyond reducing the execution time, it also encompasses the evaluation of two Java parallel APIs and the impact of parallelization on different algorithms.

Algorithm 1 High level parallel algorithm

```
1: Input: an ensemble $E$, $num\_threads$, a data stream $S$
2: $P \leftarrow Create\_service\_thread\_pool(num\_threads)$
3: $T \leftarrow Create\_trainers\_collection(E)$
4: for each arriving instance $I$ in stream $S$ do
5:     $E$.classify($I$)
6:     for each trainer $T_i$ in trainers $T$ do
7:         $k \leftarrow poisson(\lambda)$
8:         $T_i.update(I, k)$
9:     end for
10:    for all trainers $T$ do in parallel
11:        $W\_inst \leftarrow I * k$
12:        $Train\_on\_instance(W\_inst)$
13:    end for
14:    if change detected then
15:        $reset\_classifier$
16:    end if
17: end for
```

In lines 2-3, a thread pool is started, and one Trainer (runnable) is created and associated for each learner of the ensemble. For each arriving data instance (lines 4-17), votes from all the learners are aggregated to provide the predictions (line 5). Then, the *Poisson*

[1] Available at https://github.com/Waikato/moa

weights are computed, and trainers are updated for training in lines 6-9. The prediction phase has a low computational cost because the algorithm uses Hoeffding trees (DOMINGOS; HULTEN, 2000), and thus, it is carried out sequentially. On the other hand, training involves updating statistics on many nodes of the tree, calculating new splits, and detecting data distribution changes in methods that implement concept drift detection (e.g., all ensembles except the original OzaBag). As the training phase dominates the computational cost, parallelism is implemented by simultaneously training many learners (lines 10-13). Finally, lines 14-16 represent the global change detector, present on OBAdwin, where the ensemble's worst learner will be replaced with a brand new one.

5.2 Methodology

To better evaluate the efficacy of the solution, we implement it on four bagging ensemble algorithms from the MOA framework using two Java parallel APIs. The bagging algorithms are OzaBag, OzaBagAdwin, LeveragingBag and AdaptiveRandomForest. Descriptions of the bagging algorithms and Java parallel APIs are provided in section 2.2 and section 3.2, respectively. This way, we can evaluate if the proposal is really applicable to a wide array of algorithms from the bagging ensemble class. In addition, we can compare the two frameworks and weight their strengths and weaknesses to decide which one is better for the task.

Regarding the datasets, we tested the implementations on five standard ML benchmark datasets: Airlines, GMSC, Electricity, Covertype and KDD'99. A brief description of each dataset is provided in subsection 5.2.1, as they are going to be used throughout this thesis. The experiments were run multiple times to increase the confidence and all results presented use the average values.

After analyzing the execution time and speedups, we run a set of auxiliary experiments in an attempt to determine the bottleneck of the solution. In this step, we use measurements like the node count and the footprint, we investigate heap memory and garbage collector performance, and we profile the execution with the aid of specialized tools.

Specific details about the experimental setup and its results are shown in section 5.3.

5.2.1 Datasets

In the course of this thesis, six datasets were used in different combinations accross the experiments. To keep all dataset information together, we present the six datasets in here and only specify the names of the datasets used on future experimental evaluations. Five of the datasets used in this thesis are open access[2]. We provide a short description

[2] Available at https://github.com/hmgomes/AdaptiveRandomForest

of each dataset as well as a summary of their characteristics in Table 6.

Table 6 – Summary of dataset statistics

Datasets	Airlines	GMSC	Electricity	Covertype	KDD'99	Kyoto
# Instances	540k	150k	45k	581k	4,800k	725k
# Attributes	7	10	8	54	41	12
# Nominal feat	4	0	1	45	7	0
Normalized	No	No	Yes	Yes	Yes	Yes

- The regression dataset from Ikonomovska inspired the Airlines dataset. The task is to predict whether a given flight will be delayed, given information on the scheduled departure. Thus, it has two possible classes: delayed or not delayed.

- The Electricity dataset was collected from the Australian New South Wales Electricity Market, where prices are not fixed. These prices are affected by the demand and supply of the market itself and set every 5 min. The Electricity dataset tries to identify the price changes (two possible classes: up or down) relative to a moving average of the last 24h. An essential aspect of this dataset is that it exhibits temporal dependencies.

- The give me some credit (GMSC) dataset is a credit scoring dataset where the objective is to decide whether a loan should be allowed. This decision is crucial for banks since erroneous loans lead to the risk of default and unnecessary expenses on future lawsuits. The dataset contains historical data on borrowers.

- The forest covertype dataset represents forest cover type for 30 x 30 m cells obtained from the US Forest Service Region 2 resource information system (RIS) data. Each class corresponds to a different cover type. The numeric attributes are all binary. Moreover, there are seven imbalanced class labels.

- The KDD99 is often used for assessing data stream mining algorithms' accuracy due to its ephemeral characteristics. It corresponds to a cyber attack detection problem, (i.e., attack or common access),which is an inherent streaming scenario since instances are sequentially presented as a time series (GOMES et al., 2017b).

- The Kyoto dataset is an IDS dataset created by researchers from the University of Kyoto. The task is to predict if a flow is an attack of regular traffic. They used honeypots composed of many devices like servers, printers, and IP cameras, among others.

5.3 Experimental evaluation

In the experiments conducted to evaluate the parallelization of the bagging ensembles, we measure execution time as wall clock, including the prediction and the training phases of the ensembles. The experiments were made on two different platforms, in a dedicated environment. The hardware specification for both platforms is available in Table 7.

Table 7 – Hardware specifications

Processor	Xeon Silver 4208	i7-2600
Cores/socket	8	4
Threads/core	deact.	2
Clock frequency (GHz)	2.1	3.4
L1 cache (core)	32 KB	-
L2 cache (core)	1024 KB	-
L3 cache (shared)	11264 KB	8192 KB
Memory (GB)	128	16
Memory channels	6	2
Maximum bandwidth	107.3 GiB/s	21 GB/s

For the sequential execution we used the original code with no parallel framework involved. Whereas, for the parallel executions, we varied the amount of threads in the range [1-7] while using two values for ensemble size [100, 150]. The chosen ensemble sizes were based on some related work that demonstrate that the reduction in deviance asymptotes with more than 100 members in the ensemble (SAFFARI et al., 2009; PANDA et al., 2009). We still perform experiments with a higher amount of members in order to stress both the hardware and the software. We have carefully executed the experiments in an environment without external interference. We executed each parameter configuration 5 times and averaged the results using the latest MOA version[3] with the required additions and modifications for the different strategies. A new MOA package was built and used to run the experiments.

5.3.1 Results

The results from the experiments are shown in Figures 4, 5, 6, and 7 in the form of line charts. Where the first and second row display the results for experiments with ensemble size 100 and 150, respectively. Each column is assigned to one of the datasets, as indicated above the top row. All charts in the figure have the same Y-axis scale, and each chart has one series for each ensemble method as shown in the legend.

It is possible to note that the two platforms had a very different outcome regarding the actual speedup. In the i7-2600 there is usually a plateau when going past 4 threads,

[3] Available at: https://github.com/Waikato/moa

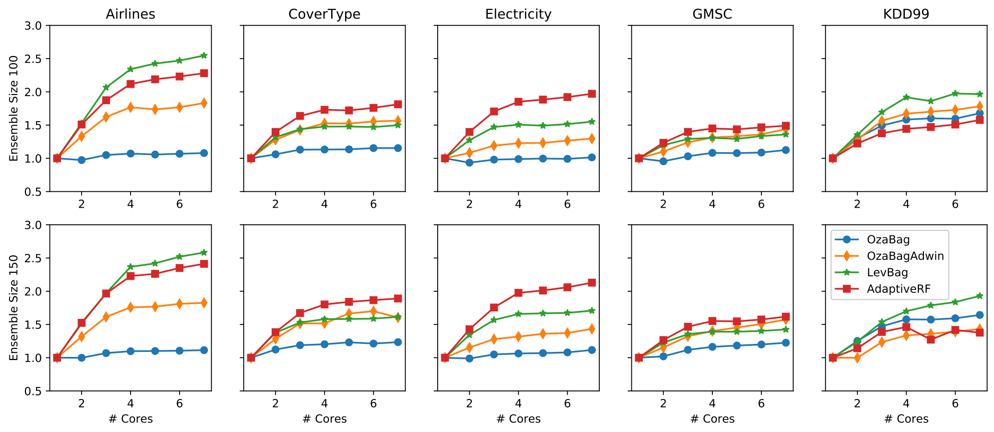


Figure 4 – Speed up results with ForkJoin framework on i7-2600.

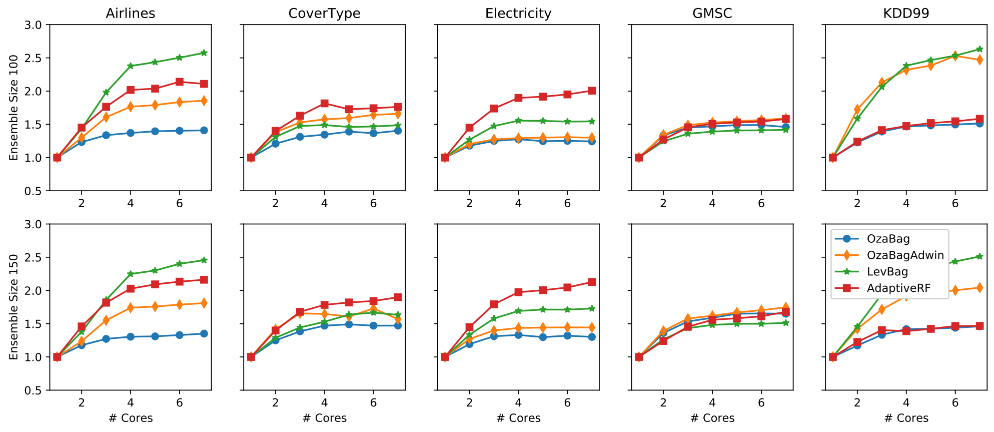


Figure 5 – Speed up results with ExecutorService framework on i7-2600.

while the Xeon 4208 has a very poor performance, presenting negative speedups in some cases.

The reason for the plateau in the i7 platform is related to hyper threading, which was designed to share the Logic and Arithmetic Unit (LAU) of a core when the processes spend a lot of time waiting for I/O operations. This application, however, is computationally intensive, which decreases the impact (i.e., performance gain) of adding extra threads with a constant problem size. This result goes hand in hand with Amdhal's Law, which states that this gradual loss of efficiency is expected when the input size of a program remains constant and the amount of parallel threads is increased.

A behavior presented in these results that persists throughout all this thesis is the smaller scalability of OzaBag (and its variants) algorithm compared to any of the other algorithms. All the reasons are related to the lower λ used in the Poisson distribution for the OzaBag algorithms. This results in a smaller weights for the instances, leading to simpler, less computationally intensive, models. The opposite is true for the non-OzaBag

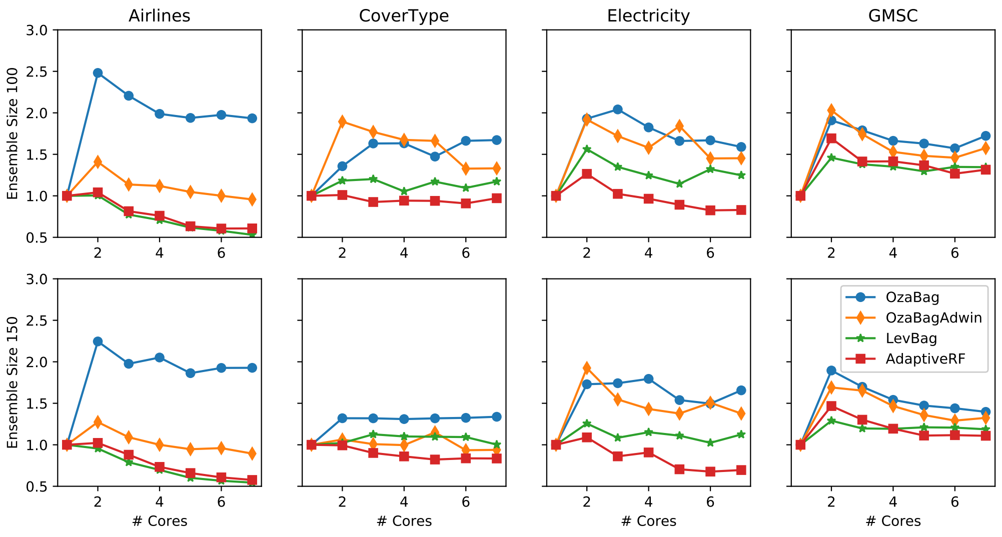


Figure 6 – Speed up results with ForkJoin framework on Xeon 4208.

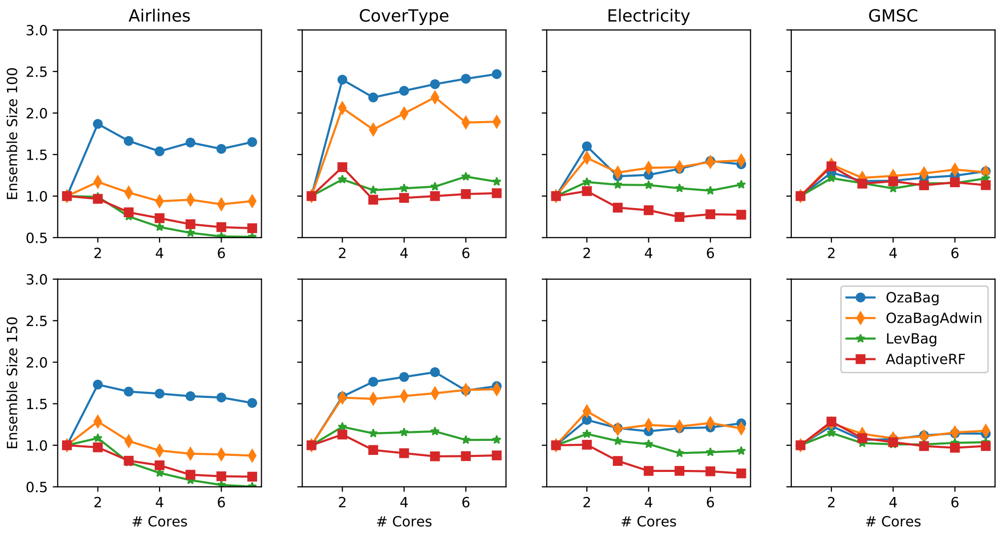


Figure 7 – Speed up results with ExecutorService framework on Xeon 4208.

algorithms, they produce bigger models that are computationally more intensive, which leads to better scalability

Regarding the parallel frameworks, there are slight differences. The most noticeable is regarding the OzaBag scalability on all datasets except KDD'99. OzaBag presents a very small scalability when implemented with the ForkJoin framework, on the other hand, its scalability is more consistent with the ExecutorService. This happens because the ForkJoin framework has a work-stealing algorithm that adds unnecessary overhead when the tasks being parallelized have similar workloads. Such is the case of the OzaBag algorithm, where the variability between the weak learners is smaller than the rest of the algorithms thanks to the smaller λ and the absence of a drift detector. The drift detector is the reason why OzaBagAdwin benefits from the ForkJoin's work-stealing, since its drift detection mechanism resets some trees and creates more variability in the workload of the tasks.

5.3.2 Node count analysis

The KDD'99 dataset presents peculiar trends, which require additional data to comprehend. The initial hypotheses is a relationship with the size of the models created by each algorithm in this dataset. Therefore, we modified the code to output the size of all the members in the ensemble (i.e., the total sum of nodes across all models) every ten thousand (10000) examples. The averages of these readings are shown in Table 8, where **bold** numbers represent the highest and *italic* the lowest value for that specific dataset (i.e., row).

Dataset	OzaBag	OzaBagAdwin	LBag	ARF
Airlines	457193	*372750*	613744	**1278406**
Covertype	9313	*389*	3835	**19134**
Electrical	2480	*615*	4361	**31488**
GMSC	*3724*	5200	33227	**187246**
KDD99	**19604**	*1645*	1767	2213

Table 8 – Mean sum of tree nodes from the whole ensemble.

Two trends are easily noticeable in Table 8. Firstly, the airlines dataset generates the biggest node count. The models generated from this dataset grow faster than the other datasets because of its nominal attributes with many possible values. They cause the tree to grow in width, since each split on any nominal attribute will create as many child nodes as the possible values of the attribute. In addition, as pointed in (Gomes et al., 2019), nominal attributes with a high amount of values will usually present a good information gain, increasing the probability of a split in such attribute. The second trend is that ARF has, usually, the biggest node count. This is a result of two combined factors: (*i*) the higher λ allows a faster growth thanks to higher resampling (shared trait with LeveragingBag); (*ii*) the random subspaces technique reduces the amount of instances needed to achieve the required confidence in the hoeffding bound to make a split, allowing the model to make splits after processing fewer examples.

The exception to the second trend happens on the KDD'99 dataset, as the highest node count is achieved by the OzaBag algorithm. Notice that although having the highest node count in all the other datasets, ARF has a very small node count with KDD'99. This is replicated by all algorithms employing a drift detection mechanism. In summary, the exception occurs because the dataset has a much higher number of instances and there are a lot of changes in data distribution, causing the drift detectors to trigger model resets very often. Since OzaBag has no drift detection its models keep growing indefinitely, leading to the higher node count. The node count can also explain why the ARF algorithm achieves its worst scalability on the KDD'99 dataset. Since the models are small and resets are triggered often, the workload never gets big enough to improve the performance through parallelism.

5.3.3 Processing time analysis

One of the objectives of this experimental evaluation, was deciding which parallel API suits our needs the best. Since the speedup is a relative measure, we present the results using the absolute processing time. The absolute value, may provide a better comparison between both APIs when using the same configuration on the i7 platform. The charts containing the processing time are shown in Figures 8, 9, 10, 11, and 12.

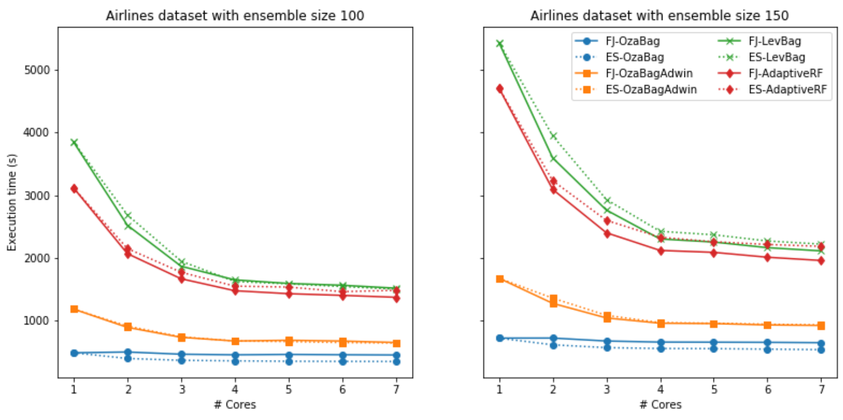


Figure 8 – Execution time of both frameworks on i7 with Airlines dataset.

In summary, the execution time does not reduce by a substantial margin after the fourth thread is added, reinforcing the behavior presented in the speedup. Also, these charts aid in illustrating the difference in algorithm complexity. Most datasets will have

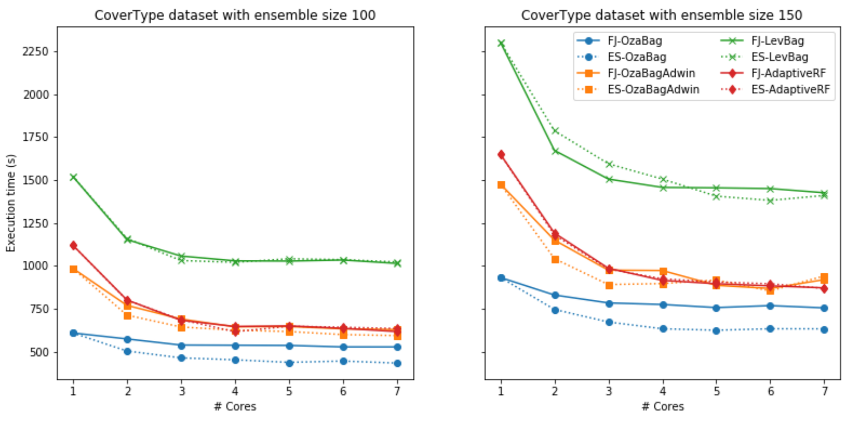


Figure 9 – Execution time of both frameworks on i7 with CoverType dataset.

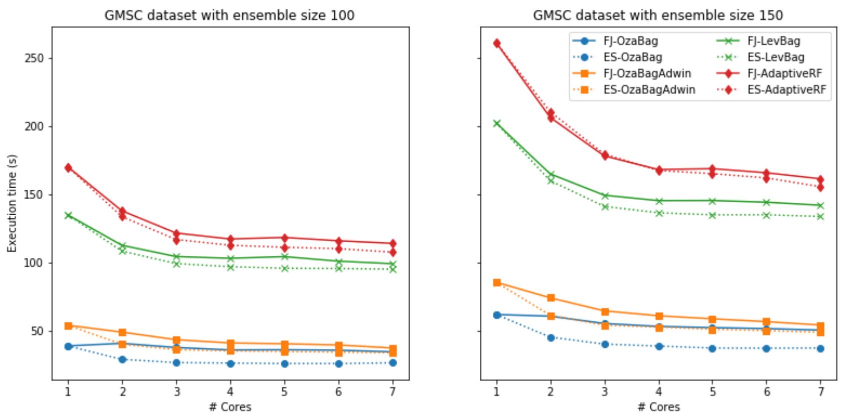


Figure 10 – Execution time of both frameworks on i7 with GMSC dataset.

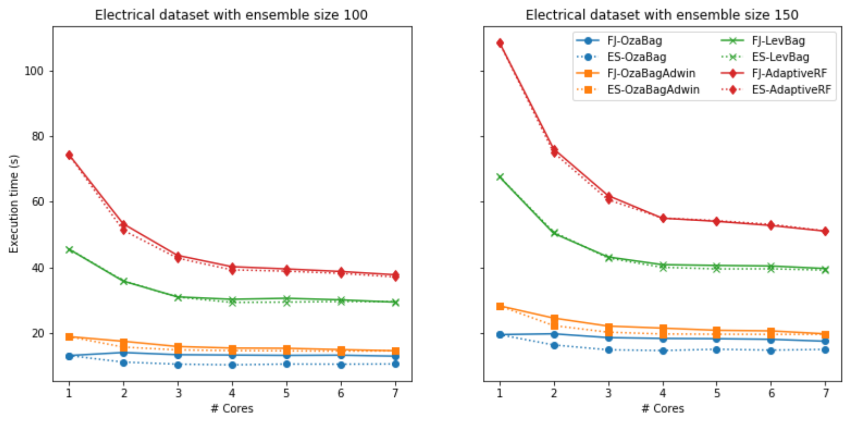


Figure 11 – Execution time of both frameworks on i7 with Electrical dataset.

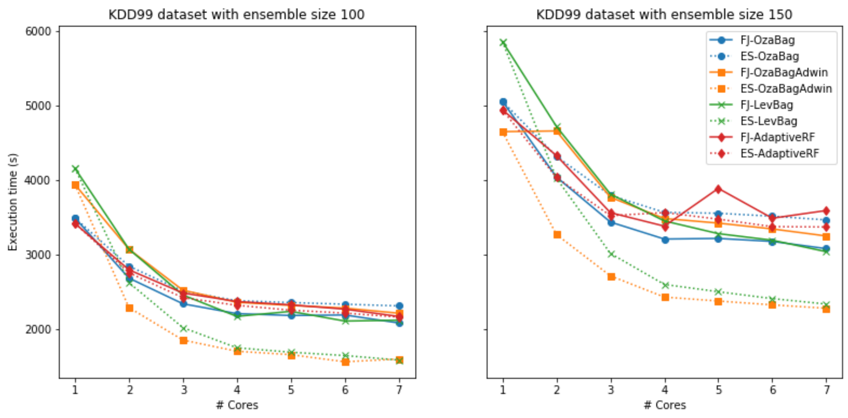


Figure 12 – Execution time of both frameworks on i7 with KDD'99 dataset.

OzaBag with the fastest execution time, followed by OzaBagAdwin and then ARF and LeveragingBag are interchangeably the slowest algorithms to complete.

It is possible to see that ExecutorService is usually better, or at least even, with ForkJoin, except on the airlines dataset and the special case of KDD'99 coupled with OzaBag. This reinforces the relationship between model diversity and the work-load algorithm from ForkJoin, since the airlines is the dataset that produces the models with the highest node counts and, consequently, diversity.

5.4 Bottleneck investigation

The poor results achieved on the Xeon platform prompted a deeper analysis of the implementation. One data that could potentially help in explaining some behaviors of the algorithms and could be collected with relatively ease, was the maximum footprint of each algorithm. We used the GNU Time[4] command to collect this data. We used only the ensembles with 100 members and allowed the JVM to use 8 GB of memory (i7-2600). The maximum resident memory for each dataset and algorithm can be seen in Table 9.

Dataset	ARF	LBag	OBAdwin	OB
Airlines	6.5GB	7.9GB	5.1GB	6.2GB
Electrical	2.1GB	1.2GB	0.24GB	0.34GB
GMSC	3.4GB	1.6GB	0.52GB	0.53GB
Covertype	2.2GB	3.4GB	5.8GB	3.3GB
KDD99	0.53GB	3.46GB	3.33GB	4.04GB

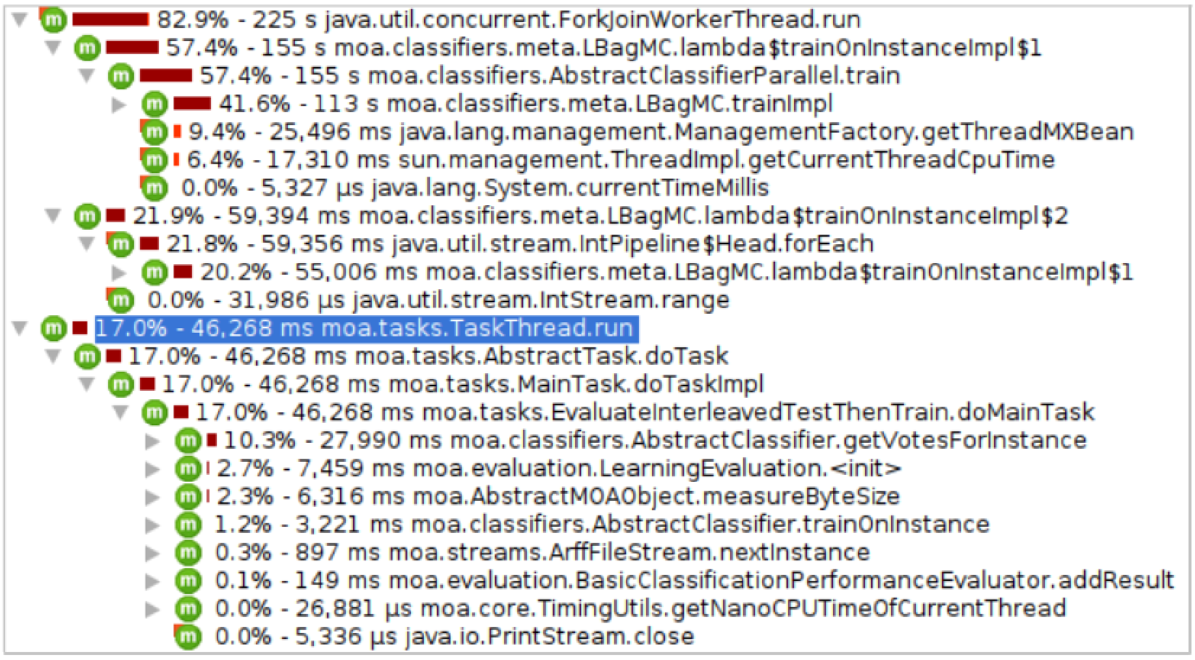
Table 9 – Maximum memory footprint (GB) for each dataset and algorithm combination for ensembles w/ 100 members.

Although the memory footprint seems related to processing time in general (e.g., airlines has the slowest processing time and the biggest footprint) it did not provide any insights on why the performance was so poor. Thus, we decided to profile the application to have a better understanding of what was happening. We used the trial version of the JProfiler[5], a Java professional 'all-in-one' profiler. We present a sample of the results using the algorithm LBag with the dataset GMSC using an ensemble size of 100 in Figures 13, 14, and 15.

In Figure 13 it is possible to see the amount of time used by each method in the code. The great majority of time is spent in the ForkJoinWorkerThread class, which is where the training happens. Based on this report, it is clear that training (82.9%) takes much more time than classifying (10.3%).

In Figure 14 it is possible to see two interfaces of the profiler. In the upper part of the figure, a detailed thread execution report is shown. Threads are identified by their

[4] https://www.gnu.org/software/time/
[5] Available at: https://www.ej-technologies.com/products/jprofiler/overview.html



```
▼ 82.9% - 225 s java.util.concurrent.ForkJoinWorkerThread.run
  ▼ 57.4% - 155 s moa.classifiers.meta.LBagMC.lambda$trainOnInstanceImpl$1
    ▼ 57.4% - 155 s moa.classifiers.AbstractClassifierParallel.train
      ▶ 41.6% - 113 s moa.classifiers.meta.LBagMC.trainImpl
        9.4% - 25,496 ms java.lang.management.ManagementFactory.getThreadMXBean
        6.4% - 17,310 ms sun.management.ThreadImpl.getCurrentThreadCpuTime
        0.0% - 5,327 µs java.lang.System.currentTimeMillis
  ▼ 21.9% - 59,394 ms moa.classifiers.meta.LBagMC.lambda$trainOnInstanceImpl$2
    ▼ 21.8% - 59,356 ms java.util.stream.IntPipeline$Head.forEach
      ▶ 20.2% - 55,006 ms moa.classifiers.meta.LBagMC.lambda$trainOnInstanceImpl$1
    0.0% - 31,986 µs java.util.stream.IntStream.range
▼ 17.0% - 46,268 ms moa.tasks.TaskThread.run
  ▼ 17.0% - 46,268 ms moa.tasks.AbstractTask.doTask
    ▼ 17.0% - 46,268 ms moa.tasks.MainTask.doTaskImpl
      ▼ 17.0% - 46,268 ms moa.tasks.EvaluateInterleavedTestThenTrain.doMainTask
        ▶ 10.3% - 27,990 ms moa.classifiers.AbstractClassifier.getVotesForInstance
        ▶ 2.7% - 7,459 ms moa.evaluation.LearningEvaluation.<init>
        ▶ 2.3% - 6,316 ms moa.AbstractMOAObject.measureByteSize
        ▶ 1.2% - 3,221 ms moa.classifiers.AbstractClassifier.trainOnInstance
        ▶ 0.3% - 897 ms moa.streams.ArffFileStream.nextInstance
        ▶ 0.1% - 149 ms moa.evaluation.BasicClassificationPerformanceEvaluator.addResult
        ▶ 0.0% - 26,881 µs moa.core.TimingUtils.getNanoCPUTimeOfCurrentThread
          0.0% - 5,336 µs java.io.PrintStream.close
```

Figure 13 – A view of the time spent on each subprocess in JProfiler.

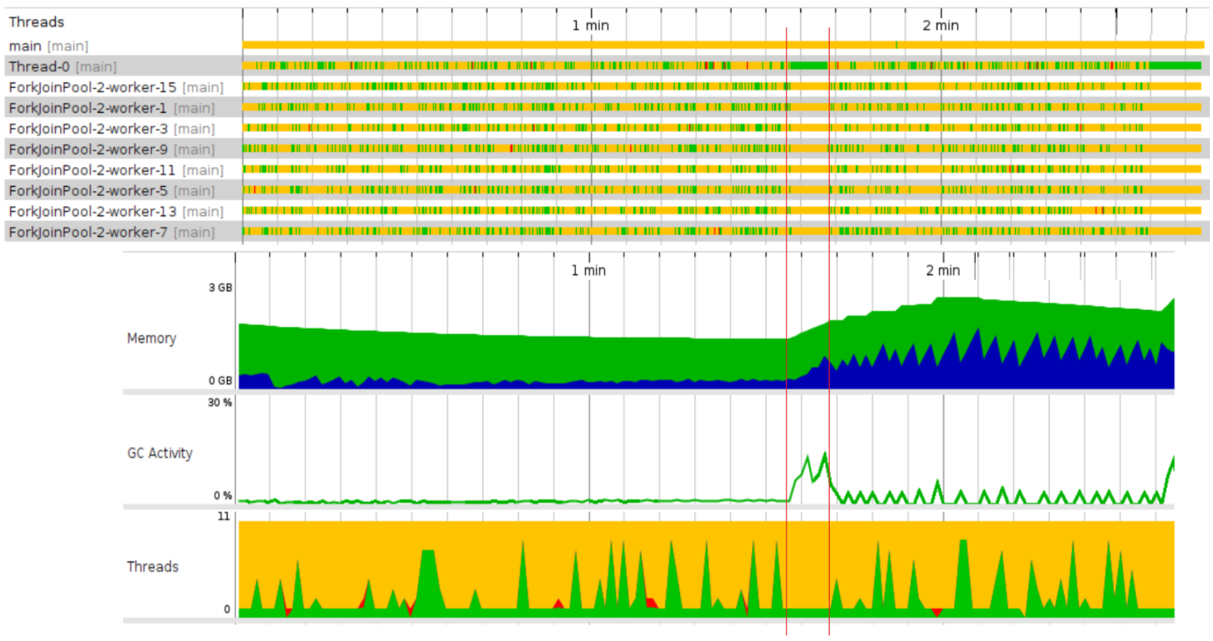


Figure 14 – A combined view of a detailed thread report (upper) and the main summary report (lower) from the JProfiler.

names on the left and their execution state is shown through colors, where green means active, yellow means waiting and red means blocked. In the lower part, one can see a simplified vision of several reports like the memory usage, the Garbage Collector (GC) activity and the Thread activity. The two red lines going across all reports were introduced as a delimiter to a specific interval where an unexpected behavior appears. The interval in question shows a spike in GC activity and memory used while the ForkJoin threads are all in waiting state. Even out of the aforementioned interval, the thread utilization is poor, as can be seen by the majority of yellow patches in both, the summarized and the detailed Thread report. Thread utilization is a clear focus point for improvement in the future.

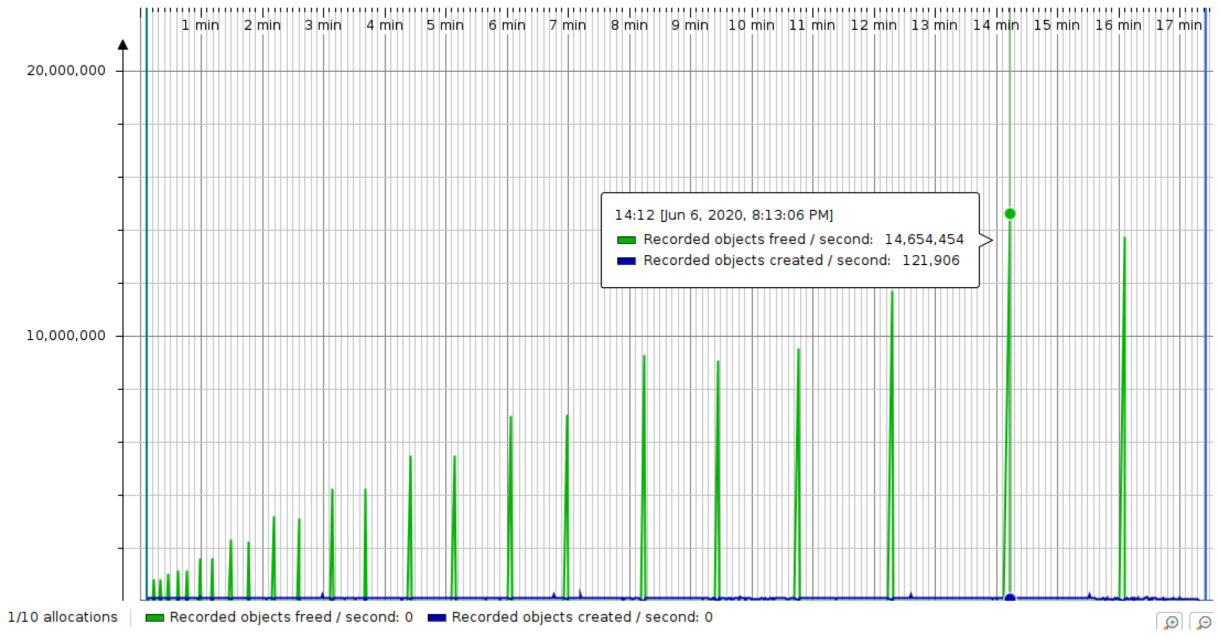


Figure 15 – A view of the Garbage Collector activity in JProfiler.

In hopes of explaining the unexpected behavior from the previous report, we rerun the profile using the same combination of algorithm and dataset, however using a more detailed memory usage collection tool. In Figure 15 a detailed GC report is shown. Although it does not show a huge spike like the previous image, the more detailed report shows several spikes, around one minute apart, that also increase in size as the time passes. This report has prompted a closer look on GC behavior.

A key concept to understand the GC behavior is the Heap memory in Java. In Figure 16 one can visualize the three generations that are used to manage the Heap. The Young Generation (YG) corresponds to data structures recently allocated, and when this space fills up a *minor* GC occurs. Minor GCs The Old (Tenured) Generation (OG) is where long surviving objects are stored, and when this space fills up a *major* GC occurs. The Permanent Generation consists of JVM metadata for the runtime classes and application methods.

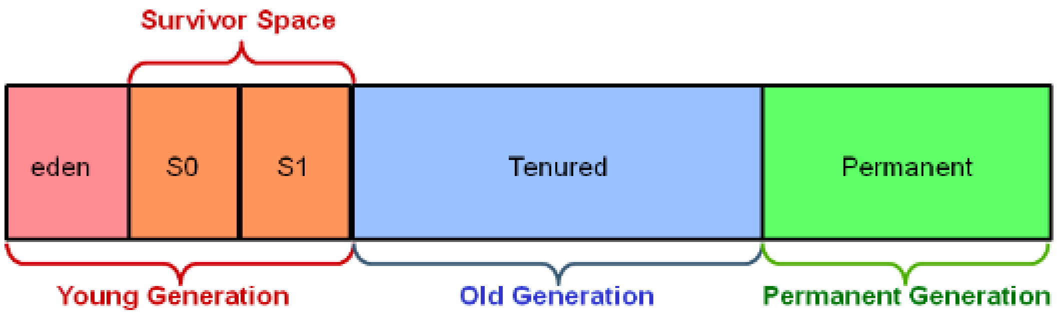


Figure 16 – Heap memory division in generations (ORACLE, 2021)

The YG is further subdivided in Eden and survivor spaces. Survivor spaces act like an aging system, where data that survive a minor GC is progressively aged. When they reach the maximum age of the YG, they get promoted to the OG. All GCs are done with the mark-and-sweep algorithm, therefore all memory pointers from the specific generation must be checked. If the maximum Heap parameter is higher than the current Heap, then the Heap is increased. If more memory is needed and the Heap can not increase, then the program aborts.

By the behaviors shown in the profiler, it seems that the YG was being stressed, as data was replicated and then discarded. One alternative was using a Parallel GC, which would not block all other Threads, however it did not improve the execution.

5.5 Summary and closing thoughts

The main findings from this Chapter can be summarized as follows.

❏ The ForkJoin API provides additional functionality that improves the performance of recursive and unbalanced tasks. In the context of ensembles, a smaller diversity among the members of the ensemble translates into a smaller variation of the workload, which does not benefit from techniques like work-stealing. Even worse, the additional work required for the work-stealing technique becomes processing overhead and deteriorates the performance.

❏ The ForkJoin API is a bad option to use with the OzaBag algorithm because Ozabag generates an ensemble with small diversity among its members. The small diversity of Ozabag is a consequence of its design, because it employs only the resampling strategy with $\lambda = 1$ without additional techniques. The small λ value means that the weights applied to the instances will possess a small range, leading to a tendency that the models will develop in a very similar manner.

❏ On the other hand, the AdaptiveRandomForest (ARF) algorithm is, theoretically, a good option to use the ForkJoin API because the ARF algorithm employs several mechanisms (e.g., higher λ value, random subsets of attributes, and individual drift and warning detectors) with the objective of improving the diversity among the members of the ensemble. Ultimately, this leads to bigger diversity in the ensemble members and consequently in the workload. However, the ForkJoin can still be the worse option even with this algorithm.

❏ The ExecutorService API provides more control to the programmer and does not have additional functionality. Both characteristics translates into a smaller processing overhead. As opposite to the ForkJoin API, ExecutorService is, theoretically, better for homogeneous workloads.

❏ Dataset characteristics are important when determining the computational cost, scalability and model behavior of the algorithms. For example, if a dataset is very large and contains a lot of changes in data distribution, OzaBag will probably present the biggest models, with the highest computational cost. The reason for such behavior is the absence of any drift detector in OzaBag, which means the models are always growing without any mechanism to reset them. On the other hand, all algorithms with drift detectors will regularly reset models, which prevents the models to grow too large.

❏ Datasets that posses nominal attributes with a high number of possible values (e.g., Airlines dataset) present a faster growth in the number of nodes of the models. This happens because each tree creates a leaf node for each possible value in the nominal attribute. Since the nodes have to be updated when training, more nodes in the model translates into a higher computational cost.

- With a much bigger instance count, KDD'99 presents atypical behavior when compared to other datasets. Although being the dataset with the most instances, the generated models present a small number of nodes for the algorithms with drift detectors. Such behavior happens because drifts are detected often, which triggers model resets.

- Based on the experiments we performed, it is safer to use ExecutorService as the default option. Even when using algorithms with several diversity enhancing mechanisms (e.g., ARF), there are cases where ExecutorService presented a better performance. This result indicates the existence of a diversity threshold in the workload that makes ForkJoin better. However, the conditions that benefit from the additional functionalities from ForkJoin are rarely met (i.e., ExecutorService presents smaller execution time in most cases shown in Figs. 8, 9, 10, 11, and 12.

- The four algorithms have different degrees of complexity. In general, LeveragingBag and AdaptiveRandomForest present the slowest processing time, the biggest models, and the best scalability. Contrarily, OzaBag and OzaBagADWIN present the fastest processing time, the smallest models and the worst scalability. The common characteristic among algorithms in both groups is the λ value, which shows the importance of the resampling strategy with regards to model complexity.

- Based on the bottleneck study, the parallel implementations suffer from poor efficiency. One aspect that needs to be improved is the poor thread utilization, as shown in Fig. 14. Memory usage should also be improved, the memory behavior shows massive bursts of creation and deletion of objects, which is detrimental to performance.

Based on these findings, it was clear that a task-parallel solution, independently of the parallel API used, was not capable of achieving speedups of at least 3x with 8 cores. Thus, further improvements in efficiency are required.

Chapter 6

Introducing mini-batching to data stream ensembles

Although a task-parallel implementation on ensembles looks straightforward, its performance can be severely hindered by poor memory usage as shown in section 5.4. For instance, high-frequency access to data structures that are larger than cache memory can raise performance bottlenecks. Also, algorithms that continuously perform memory allocation/release operations to discard old models and create new ones during the learning and training process may pressure the garbage collection. In fact, a better memory usage can improve the implementation by itself, even when running the ensemble sequentially. A different approach, that groups several data instances of a stream in a mini-batch can help mitigate such problems. Formally, a Mini-batch B is a group of s data elements $x_1, x_2, ..., x_s$ arriving through a data-stream, that is, $B = \{x_i\}_{i=1}^{s}$. When the stream is terminated, B may be smaller than s. We propose a mini-batching strategy for data-stream ensembles, capable of improving the memory locality in Algorithm 2.

In Algorithm 2, it is possible to note that the first command of the main loop (from line 2 to 23) changes from the classification of the arriving instance (in Alg. 1) to the append operation that groups arriving instances in the mini-batch (line 3). Then, when the mini-batch size is equal to the desired size L_{mb} passed as argument (line 4), the mini-batch is computed. The most intuitive manner to compute the mini-batch would be to iterate through the mini-batch in the outer loop, while the inner loop stays the same (i.e., iterates over the classifiers of the ensemble). However, iterating the classifiers in the outer loop (lines 5 and 11 for classification and training, respectively) and the mini-batch instances in the inner loop (lines 6 and 12 for classification and training, respectively), is a more efficient alternative. In line 10 the votes from all classifiers are aggregated, and

Algorithm 2 sequential mini-batching algorithm

```
1: **Input**: an ensemble $E$, a data stream $S$, mini-batch size $L_{mb}$
2: for each arriving instance $I$ in stream $S$ do
3:     $B.append(I)$
4:     if $B.size() == L_{mb}$ or $S.has\_ended()$ then
5:         for each classifier $C_i$ in ensemble $E$ do
6:             for each instance $I$ in $B$ do
7:                 $votes_i.append(C_i.classify(I))$
8:             end for
9:         end for
10:        $E.aggregate(votes)$
11:        for each classifier $C_i$ in ensemble $E$ do
12:            for each instance $I$ in $B$ do
13:                $k \leftarrow poisson(\lambda)$
14:                $W\_inst \leftarrow I * k$
15:                $C_i.train\_on\_instance(W\_inst)$
16:            end for
17:            if change detected then
18:                $reset\_classifier$
19:            end if
20:        end for
21:        $B.clear()$
22:    end if
23: end for
```

the algorithm proceeds to the training phase. Lines 13 to 15 show the basic operations made for each instance of the mini-batch for each classifier of the ensemble. Line 19 shows the change detection taking place. In this instance, the position the change detection is presented refers to all ensembles tested except OzaBag and OzaBagAdwin. OzaBag has no change detection and OzaBagAdwin performs a global change detection. Finally, the mini-batch is emptied in line 21 and the process of accumulating instances restart. This loops repeat until a finishing condition arrives.

In essence, the memory locality of the algorithm is improved by grouping instances and changing the loop order. This happens because each classifier can process the mini-batch uninterruptedly. Specifically, it allows the reuse of classifier data structures that have already been brought to the higher levels of memory hierarchy.

Notice that each prediction model usually is several times larger than a single data element, and the size of the whole ensemble can be (almost always) several times larger than the cache memory. Thus, the memory access cost of one model is significantly higher than the memory access cost of one data element. Without mini-batching, Algorithm 1 reuses each (low memory cost) data element which is processed by all (high memory cost) models, strengthening the memory bottleneck. Instead, the mini-batching strategy loads each prediction model only once and reuses it as many times as the mini-batch size parameter (L_{mb}).

6.1 Improvement in sequential performance

To demonstrate the improvement in performance caused by a better memory locality, we performed some experiments using mini-batching with a sequential algorithm. We performed experiments with ensembles of 100 members, using all six algorithms and four datasets. The machine used in this experiments was the Xeon 4208. Results are presented in Figure 17

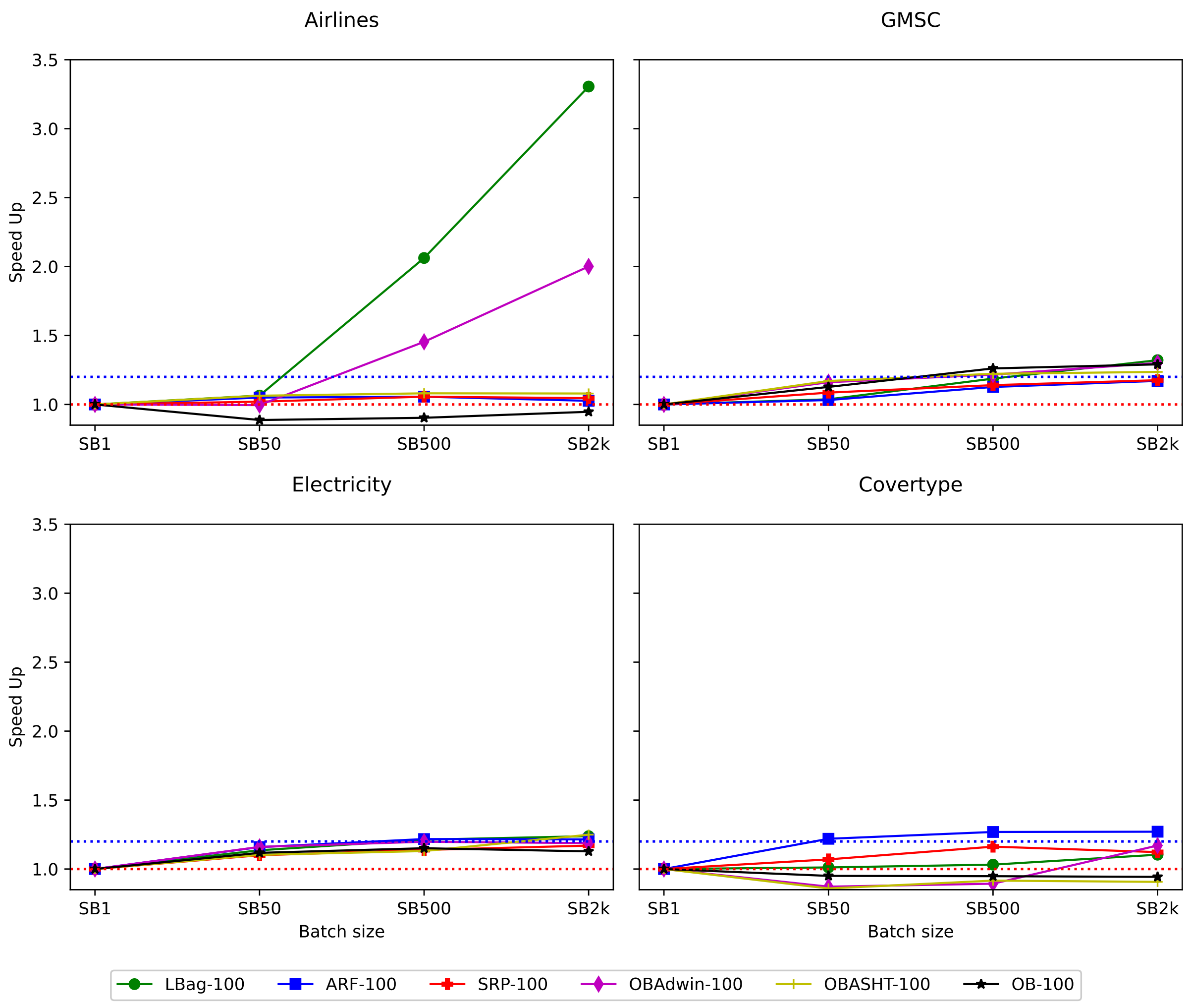


Figure 17 – Sequential ‘Speedup’ with mini-batching.

Figure 17 shows one series for each algorithm, and one chart for each dataset. each chart is scaled from 1 to 3.5. Each chart has two horizontal lines a red one is the line $y = 1$ and the blue one is $y = 1.25$. These lines are to be used as guides that show when an implementation performed worse than baseline (SB1) or at least 25% better than baseline, when introducing the mini-batching straetgy. The airlines dataset shows impressive results for LeveragingBag and OzaBagAdwin. On GMSC and Electricity datasets, algorithms are around 25% better, just by adding mini-batching, even without parallelism. Airlines and Covertype shows more mixed results.

6.2 Reuse distance analysis

To show that mini-batching improves memory locality, we present experimental results based on the reuse distance measure proposed by (YUAN et al., 2019). Reuse distance can be efficiently obtained by instrumenting the application and directly relates to cache performance. In addition, the reuse distance histogram is a compact summary of the application's memory locality, since it is related to the miss ratio of the cache. In essence, the behavior of the frequency of large reuse distance is mirrored by the cache-misses, which, when decreased, provides a better performance.

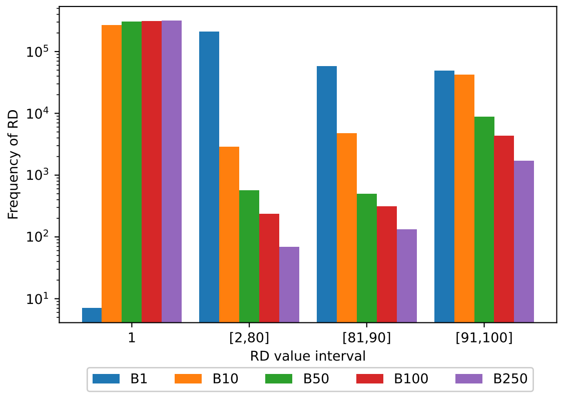
To collect the required data, we instrumented the ensemble code to track the order each classifier of the ensemble was accessed. We also limited the input stream to $n = 5000$ elements, and used an ensemble of $m = 100$ classifiers. To show the impact of the mini-batch size in the RD, we varied the mini-batch size b with $[1, 10, 50, 100, 250]$ values.

As it was explained in section 2.2, the parameter λ of the Poisson distribution affects the weight each instance receives in the process called re-sampling. If the algorithm uses $\lambda = 1$, over 30% of the instances will not be used to train the classifiers. This characteristic can influence the RD positively, since a skipped instance by a classifier means that its data structures will not be loaded to the cache memory. At this point, we can group the algorithms in two categories regarding the parameter λ. The first category comprises the algorithms LBag, ARF, and SRP that use $\lambda = 6$, and the second category includes OB, OBAdwin, and OBASHT and uses $\lambda = 1$. We reported only the behavior of one representative for each group (LBag and OB), since the members of each group presented a very similar behavior.

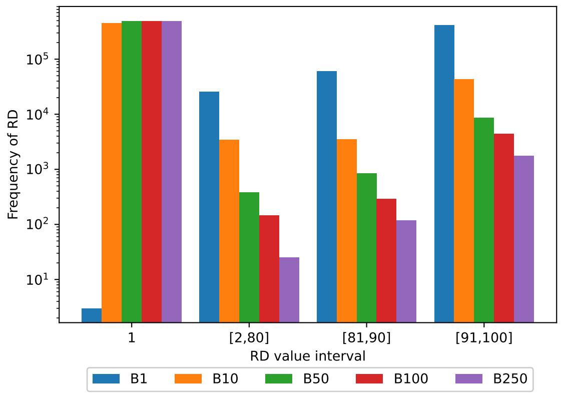
The results are shown in Figure 18, where the color of each a vertical bar is associated with the mini-batch size. The X-axis ticks show the RD value interval, while the Y-axis shows the count of the frequencies of a given interval in log scale. The RD value interval in the X-axis means that the RD for a given data is in that interval. For example, the frequencies of RDs valued between 2 and 80 are summed and the resulting frequency is the value of the vertical bar. We separate the RD frequency 1 and highlight the RD frequencies higher than 80 because they are these values and intervals are the ones that change the most when introducing mini-batching..

Both cases shown in the figure present a very similar behavior. When using mini-batch, the great majority of the RDs are 1, which means that the algorithm reuses the most recently used data. The remaining RDs are inside the high intervals, as expected given the amount of data structures (i.e., ensembles).

Notice that the larger the mini-batch size, the less frequent the high RDs are and



(a) RD histogram for OzaBag.



(b) RD histogram for LeveragingBag.

Figure 18 – RD histogram from one representant of each group (according to λ)

the higher the frequency of RD=1. For instance, near 83% of the RD for the parallel implementation falls in the range [91,100]. With mini-batches of size 10 (B10), high RDs' frequency decreases to near 8.6%. With the largest mini-batch size (n=250), only 0.35% of the RD falls into the range of [91,100]. Thus, we can conclude from this experiment that the larger the mini-batch, the fewer cache misses and the better performance. In addition, the difference between the smaller and bigger mini-batch sizes is negligible compared to the changes resulting from introducing the mini-batching approach.

6.3 Impact of mini-batching on predictive performance

Mini-batching impacts the behavior of ensembles in two ways. First, it slows down the training phase. In pure stream processing (without mini-batching), every data instance is classified and then used to train the model, which implies that every new arriving instance is classified with the most up-to-date model. On the other hand, when mini-batch is introduced, all the data instances of the current mini-batch are classified before the model is trained (as illustrated by Algorithm 2). This method reduces each algorithm's opportunities to update the models, leading to slower growth of the trees and keeping models on a more generic (less accurate) state for longer periods of time given a big enough mini-batch size.

A second impact of mini-batching on the performance of ensembles is to delay the detection of changes in data distribution (concept drifts). Unlike the pure stream method that checks for drift after processing every instance, when using mini-batch the check is made only after the whole mini-batch is processed. This effectively reduces the number of times the algorithm will check for drifts, and also group the drift detections together when more than one occurs inside the same mini-batch. As the consequence, the reaction time for data distribution changes will be slower, and, thanks to that, the predictive performance could be potentially lower.

Next, we present two additional experiments with the goal to understand and provide empirical data about the downsides of introducing the mini-batching strategy in streaming ensembles.

6.3.1 Impact of mini-batching on recall and precision

To measure the impact of mini-batching on the predictive performance, we carried out experiments with several mini-batch sizes (25, 50, 100, 250, 500, 1000, 2000) and measured the predictive performance using *precision*, *recall*, and (when possible) the number of concept-drifts detected. We used the same instance weights as the sequential version, ensuring that the only difference is the delay caused by the mini-batching. We used ensembles with 100 learners. Figure 19 shows the precision and recall measures for each combination of dataset and algorithm as we increase the mini-batch size.

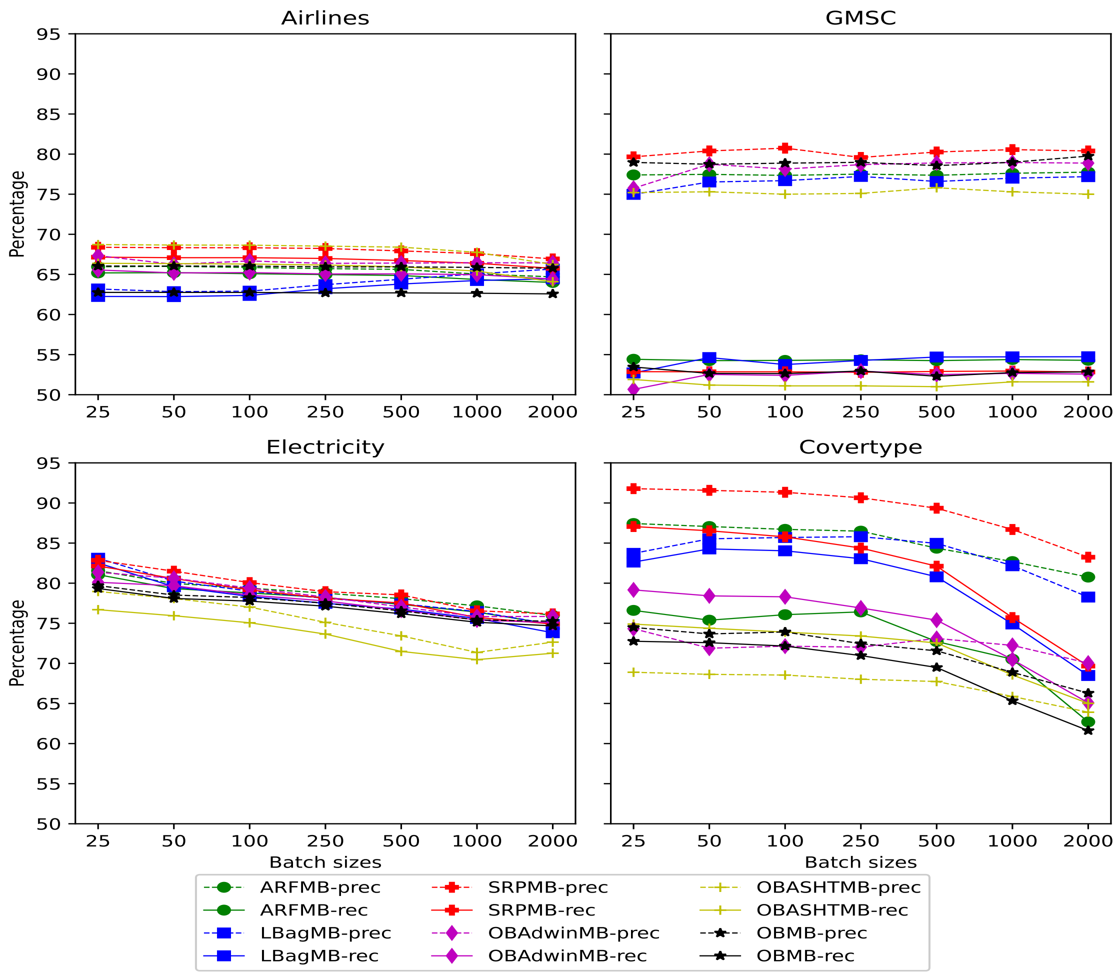


Figure 19 – The *precision* and *recall* measures for each algorithm grouped by datasets. The Y-axis shows predictive performance as percentage value. The X-axis shows the mini-batch size. Solid lines are used for *recall*, and dashed lines for *precision*.

We can observe two distinct behaviors. The increase in the mini-batch size has low impact on predictive performance for the datasets Airlines and GMSC. In contrast, a more significant decrease in predictive performance occurs with Covertype and Electricity. Results show that the impact on the predictive performance is more influenced by the dataset characteristics than by the algorithms. Nevertheless, in cases where the mini-batching strategy impacts the predictive performance, the impact tends to grow as we increase the mini-batch size.

6.3.2 Impact on change detection

Additional experiments using LBag and OBAdwin were carried out to track the number of changes detected in each dataset. As a general remark, the number of changes detected decreases as the mini-batch size increases. Small mini-batches (i.e. less than 50 instances) deviate from this behavior, as the algorithms detect more changes than in the baseline.

The spike in the number of changes is caused by the lower accuracy, which is related to the slower pace that the models are trained. The behavior described can be viewed in Figure 20 and they can help explain some prediction results from Figure 19.

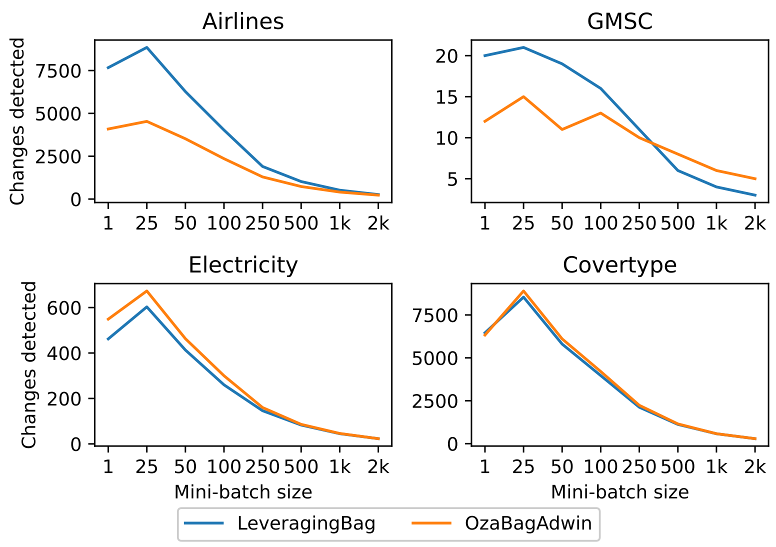


Figure 20 – Changes detected by algorithms LeveragingBag (LBag) and OzaBagAdwin (OBA) for each dataset according to the mini-batch size.

Dataset GMSC shows a minimal amount of changes detected, indicating that data distribution is stable and consistent. The consequence is that the ensemble models are rarely replaced, as only 20% of the ensemble is replaced over the full length of the 150,000 instances. On the other hand, the electricity dataset presents over 600 changes detected for the same ensemble size (100) and only 45,312 instances, which means the whole ensemble could be replaced 6 times. This indicates that electricity dataset has many data distribution changes. Therefore models become obsolete quicker and need to be replaced. This behavior explains why GMSC has a very stable predictive performance while electricity suffers a drop on all algorithms as the mini-batch size increases.

The number of changes detected is similar in airlines and covertype datasets. However, the airlines dataset has a constant predictive performance while covertype suffers a drop. One difference between both is that the airlines dataset is a binary class dataset while the covertype dataset is multiclass. Another difference is that the airlines dataset has two nominal attributes with many values, which tend to trick the learner into doing splits on them, as shown in (Gomes et al., 2019). The mini-batch impact in prediction is minimal in the airlines dataset because the initial prediction is already low. On the other hand, the covertype dataset's impact is more noticeable because the original prediction (in the incremental setting) is better and deteriorates with mini-batches.

Figure 20 shows an increase in changes detected for small batches up to 25 instance. This behavior happens in the beginning when the models are untrained and thus have a high error rate, triggering change detections more often. As the stream progresses, the models grow and become more capable of recognizing the classes, thus improving this behavior. This is alleviated in the bigger mini-batch sizes because there are a lot fewer opportunities to trigger change detections.

6.4 Summary and closing thoughts

The main findings from this Chapter can be summarized as follows.

❑ Mini-batching improves data locality of the ensembles, by reusing the models (bigger data structure) to process multiple instances. In this process, mini-batching prevents multiple fetches from the primary memory and uses data already located in the caches.

❑ Even without the parallelization, mini-batching alone is capable of improving performance by 20% in many cases. These gains are possible thanks to the change in the order of operations, which prevents the fetching of the models from the primary memory to process only one data instance.

❑ A better data locality is confirmed by a better Reuse Distance measure. As shown in the Reuse Distance histogram, with mini-batching the reuse distances with value 1 are the most frequent. However, this instrumentation used an assumption that the whole classification model could fit in cache memory, which is unlikely but the best case possible.

❑ Mini-batching can impact the predictive performance. Although algorithm characteristics are important to determine the baseline predictive performance, the behavior of the predictive performance follows a trend according to the dataset. In other words, every tested algorithm behaves the same way when classifying a given dataset.

❑ It is not clear which dataset characteristics are the most relevant to determine the intensity of the losses in predictive performance. However, it is clear that, when the deterioration happens, its intensity increases toghether with the mini-batch size.

In summary, the results presented in this Chapter demonstrate that a mini-batching strategy is capable of improving the memory locality of ensembles. In addition, the introduction of mini-batching provides a bigger performance gain than the increase of its size. The downside of the mini-batching strategy is related to the predictive performance. The mini-batching impact in predictive performance varies, mainly, according to the dataset and can be as low as less than 1% and as big as more than 10%. In smaller mini-batch sizes, the impact is almost negligible.

Chapter 7

Combination of parallelism and mini-batching for data stream ensembles

By combining both the mini-batching and the task-parallel strategies, the algorithm can achieve better performance thanks to the benefits of parallalelism and improved memory locality. The combination of both techniques provide a “bigger than the sum” result, since they increase each other’s efficiency. With this, the overall objective of improving performance of bagging ensembles in a data stream setting and adhering to this setting’s constraints was achieved. Speedups can reach almost 5X in most algorithms, with an special case where it reaches 12X.

This chapter describes the algorithm, details the experiments and presents the results obtained after the implementation of the complete solution on MOA framework and testing in three different platforms with four datasets.

In Algorithm 3 we present the unified solution, where the algorithm uses the mini-batching technique in a task-parallel model. The mini-batching is formed in the sequential portion of the code. After the mini-batch is complete, the algorithm launches the parallel tasks that will process the mini-batch. Each learner is mapped to a single task that will be scheduled by the parallel API to run in the number of available physical cores (parameter passed as argument).

The difference between Alg. 3 and 2 is the addition of the parallel framework, since the creation of the mini-batch and the order of operations is the same as in Alg.2.

In Algorithm 3, the main loop has almost the same structure as in Alg.2. The algorithm appends arriving instances in the mini-batch (line 5). Then, the mini-batch is computed

Algorithm 3 parallel mini-batching algorithm

```
 1: Input: an ensemble $E$, $num\_threads$, a data stream $S$, mini-batch size $L_{mb}$
 2: $P \leftarrow Create\_service\_thread\_pool(num\_threads)$
 3: $T \leftarrow Create\_trainers\_collection(E)$
 4: for each arriving instance $I$ in stream $S$ do
 5:     $B.append(I)$
 6:     if $B.size() == L_{mb}$ or $S.has\_ended()$ then
 7:         for each trainer $T_i$ in trainers $T$ do in parallel
 8:             $T_i.instances \leftarrow B$
 9:             for each instance $I$ in $T_i.instances$ do
10:                 $votes_i.append(T_i.classify(I))$
11:             end for
12:         end for
13:         $E.aggregate(votes)$
14:         for each trainer $T_i$ in trainers $T$ do in parallel
15:             for each instance $I$ in $T_i.instances$ do
16:                 $k \leftarrow poisson(\lambda)$
17:                 $W\_inst \leftarrow I * k$
18:                 $T_i.train\_on\_instance(W\_inst)$
19:             end for
20:             if change detected then
21:                 $reset\_classifier$
22:             end if
23:         end for
24:         $B.clear()$
25:     end if
26: end for
```

once its size reaches the desired size L_{mb} passed as argument (line 6). The logic for iterating through mini-batch instances is even more straight forward in this version. Inside each task the only iteration possible, and needed, is the iteration over the mini-batch instances, since each learner is mapped to a single task. All tasks run in parallel (lines 7 and 14). Besides the accumulation of instances to form the mini-batch, the aggregation of votes also happens in a sequential manner (line 13). The change detector has the same behavior differences as before. After processing the mini-batch, the algorithm clears the 'current' mini-batch to begin accumulating the next one (line 24.

7.1 Experimental evaluation

To evaluate the benefits of parallel mini-batching for data stream bagging ensembles, we implemented the strategy in MOA and tested its performance on several ensemble algorithms. The datasets are the same as in the previous experiments: Airlines, GMSC, Electricity and Covertype. We have increased the number of algorithms to six bagging ensembles, as follows: OzaBag, OzaBagAdwin, OzaBagASHT, LeveragingBag, AdaptiveRandomForest, and StreamingRandomPatches. We kept the same ensemble sizes as in previous experiments are also the same as the next: 100 and 150. Lastly, we used three different machines to execute the experiments, their specifications are shown in Table 10.

Table 10 – Hardware specifications

Processor	Xeon 4208	E5-2650	AMD 7702
Cores/socket	8	10	64
Clock frequency (GHz)	2.1	2.3	2.0
L1 cache (core)	32 KB	32 KB	32 KB
L2 cache (core)	1024 KB	256 KB	512 KB
L3 cache (shared)	11264 KB	25600 KB	262144 KB
Memory (GB)	128	384	1024
Memory channels	6	4	8
Maximum bandwidth	107.3 GiB/s	51.2 GB/s	204.8 GB/s

We evaluate the performance of three algorithms, with the important note that one of the three has three different configurations. That leads to five different series in the charts: a sequential implementation (baseline), a parallel implementation without mini-batching (B1), the parallel implementation with mini-batches with three different mini-batch sizes: 50, 500, and 2000. The three configurations of the parallel implementation with mini-batches are called B50, B500 and B2k. All parallel implementations were executed with 8 threads pinned to the processing cores. In spite of using all available cores on the three platforms, we chose to use the same amount of threads. This decision was made to allow the comparison of the performances as a way to validate our prototype in regards to different cache memory characteristics. However, it does not exclude the importance of a scalability study. In fact, studying the scalability using more cores is an interesting future work that should be performed to check how our prototype can adapt to bigger environments.

7.2 Performance analysis

The comparison of the five different versions of the algorithms is presented on Figure 21 using the speedup metric. Speedup was calculated compared to the baseline, and, as stated above, all parallel executions are run with a threadpool of 8. Further description on the organization of the charts is provided in the caption.

Results show that pure parallelism (without mini-batching) is erratic, yielding worse performance than sequential in many configurations. The slower execution time in the B1 version compared to the baseline reinforces the lack of efficiency when processing the examples incrementally (i.e., classifying and training a single example for the whole ensemble at a time), as demonstrated in Chapter 6.

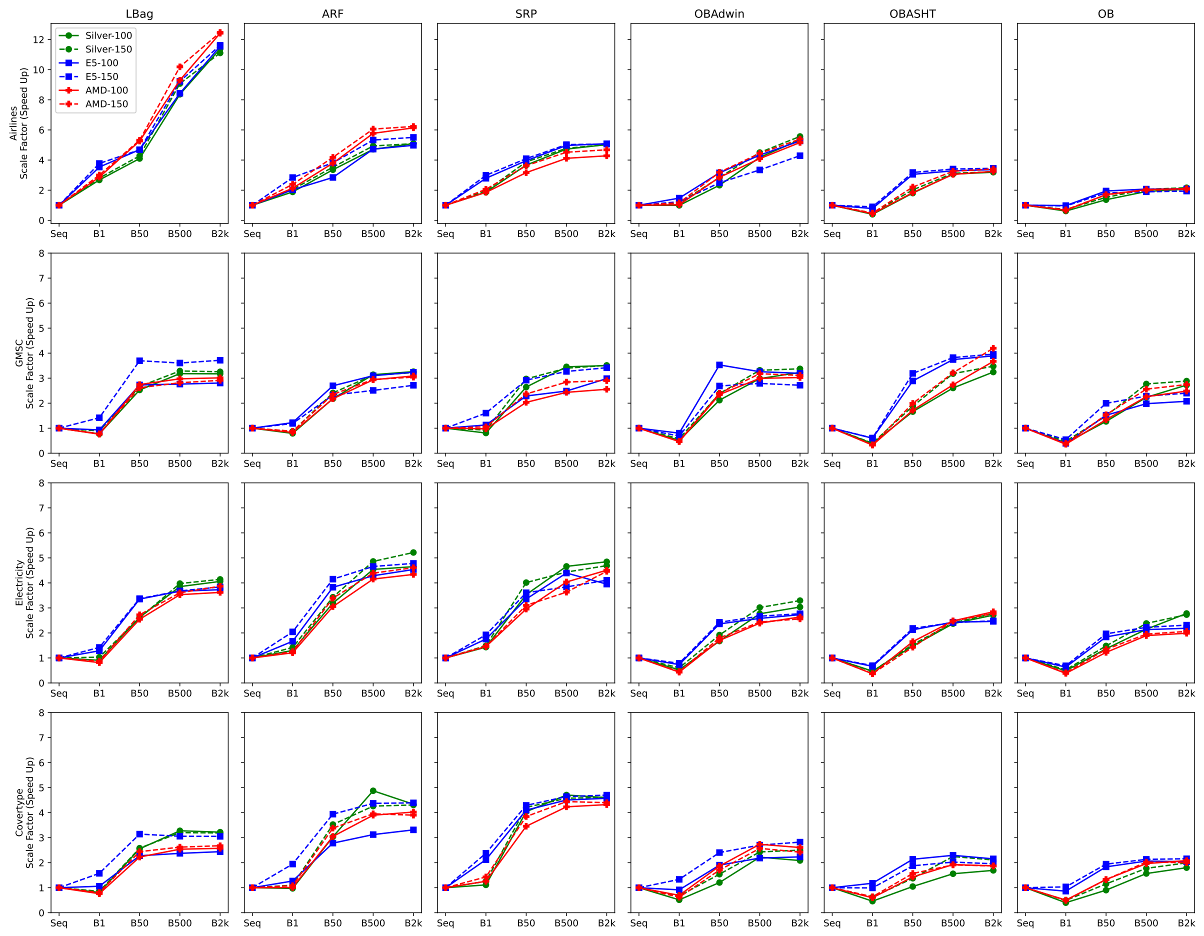


Figure 21 – **Speed up on all platforms**. Algorithms are placed on columns, while datasets are placed in different rows of the grid. Suffix 100 and 150 indicate the size of the ensemble, and are represented with solid and dashed lines, respectively. First row Y-axis is scaled to 12, every other row is scaled to 8. Algorithm implementations: (*i*) Baseline (sequential) (**Seq**), (*ii*) Parallel (**B1**), (*iii*) Parallel with mini-batches of 50 instances (**B50**), (*iv*) Parallel with mini-batches of 500 instances (**B500**), (*v*) Parallel with mini-batches of 2,000 instances (**B2k**).

In contrast, performance gains are obtained by combining parallelism with mini-batching due to better memory access patterns. Most experiments present a big leap in performance when introducing the parallel mini-batching of 50 instances (B50). However, with the exception of the airlines dataset and LBag algorithm, the performance gains decrease as we increase the mini-batch size. The decreasing performance gains behavior reinforces the results presented on the RD study in section 6.2, where the difference in RD after the initial implementation of mini-batch does not change by a big margin.

Also, speedups are closely related to the models' computational complexity, which varies according to the algorithm and dataset used. Cheaper algorithms (e.g., OzaBag and OzaBagAdwin) show lower Speedups due to smaller amounts of work per thread. As a final general note, results indicate that the solution is platform independent, since every chart presents a very similar behavior for all platforms and ensemble sizes.

An interesting result arises from the LeveragingBag algorithm when classifying the airlines dataset. Although an ideal speedup should approach 8, which is the number of cores used, the chart shows a 12X speedup in this particular combination. This superlinear speedup indicated a better synergy, however more instrumentation and extra experiments would be needed to provide a clear cause for such behavior. Since the memory-locality has been the key improvement to present good results, one logical roadmap is to collect and study real memory statistics such as cache-misses.

7.3 Cache study

We ended the last section with an interesting superlinear speedup, however, we could not explain why this has happened. Although the theoretical RD calculations from section 6.2 provide a good hint to the performance bootleneck, it can not be used to explain the superlinear speedup. Since the superlinear speedup happens on only one combination, we should find out what differences this particular case has in comparison to the others. A logical way is to collect and study real memory statistics, as opposed to a simplified theoretical model. To achieve this goal, real cache usage data should be collected during runtime. Thus, we re-executed all configurations while collecting cache data with the Linux Perf tools[1]. The experiments were performed with ensembles of 100 learners, and the results are shown in Table 11. We present the two metrics used to evaluate cache usage:

- ❏ *Cache-references*: accounts for data requests missed in the L1 and L2 caches. Whether they miss the L3 is irrelevant in this case;

- ❏ *Cache-misses*: represents the number of memory access that could not be served by any of the cache levels, therefore having to fetch data from the main memory.

Results confirm the claim that mini-batching improves the memory accesses locality, since they show the reduction a real memory statistic (e.g., cache-reference) when introducing the mini-batching technique. Although the two measures change according to dataset and algorithm characteristics, both tend to decrease with mini-batching and larger batch sizes. For the Electricity and Covertype case, the *cache-refer* starts to rise with mini-batches of 2000 instances, suggesting the existence of an optimal mini-batch size for each case.

The results from LeveragingBag on airlines show a singular behavior, replicated to a lesser extent by the OBAdwin algorithm. While most combinations present small variations on cache-misses (i.e., the first column), LBag on airlines reduces the metric by more

[1] <https://man7.org/linux/man-pages/man1/perf.1.html>

Table 11 – Measures of cache use for ensembles with 100 learners (Millions)

Algorithm	MB size	Airlines		GMSC		Electricity		Covertype	
		cache-miss	cache-refer	cache-miss	cache-refer	cache-miss	cache-refer	cache-miss	cache-refer
ARF	1	40,171	94,910	2,518	11,366	882	4,490	12,652	65,321
ARF	50	41,634	63,303	2,499	4,825	821	2,323	13,325	23,201
ARF	500	42,321	62,185	2,162	4,394	742	2,040	12,315	21,246
ARF	2000	42,522	61,728	2,047	4,369	728	2,293	12,367	21,912
LBag	1	45,337	99,010	2,600	8,962	508	2,870	17,809	104,735
LBag	50	49,425	74,854	1,680	3,746	516	1,706	16,560	47,024
LBag	500	26,659	37,783	1,645	3,497	473	1,457	19,315	45,322
LBag	2000	19,556	26,152	1,546	3,714	463	1,309	21,342	48,600
SRP	1	45,135	110,900	5,543	18,487	2,105	7,520	65,157	172,089
SRP	50	46,647	68,340	5,285	8,682	1,999	3,892	61,763	97,867
SRP	500	45,973	67,255	4,781	7,750	1,952	4,296	60,210	95,699
SRP	2000	45,973	66,395	4,559	7,912	1,863	3,916	60,906	99,117
OBASHT	1	4,779	39,986	531	4,399	225	1,714	5,927	101,370
OBASHT	50	3,918	10,629	399	1,262	171	781	5,286	40,059
OBASHT	500	3,810	9,953	353	1,033	157	717	4,648	36,992
OBASHT	2000	3,579	9,603	334	1,090	155	761	4,302	39,074
OBAdwin	1	26,627	71,987	723	5,770	232	2,037	5,780	108,281
OBAdwin	50	20,338	30,542	439	1,539	183	910	4,687	37,948
OBAdwin	500	15,417	21,888	419	1,357	177	872	5,576	35,341
OBAdwin	2000	11,669	16,414	371	1,427	149	915	6,228	33,759
OB	1	9,423	27,560	981	5,580	221	1,864	11,314	94,976
OB	50	9,810	13,606	635	1,853	180	735	9,683	36,822
OB	500	9,504	12,468	421	1,531	173	738	7,983	32,385
OB	2000	8,965	12,299	353	1,386	155	793	7,141	32,146

than 50%. In addition, the reduction on cache-refer is bigger than in the other combinations, and, as opposed to the 'default' behavior of stabilizing, it continues reducing as we increase the mini-batch size. This combination of factors leads to a much improved memory locality. It is still hard to point the cause for the better performance, however, based on the results, the LBag algorithm has a higher reuse of data structures. As stated before, the scope of this thesis is a generic solution for a whole class of algorithms. Thus an in-depth study of a single algorithm falls out of the scope.

7.4 Summary and closing thoughts

The main findings from this Chapter can be summarized as:

❏ We have built upon previous results and presented empirical proof that mini-batching can greatly improve the memory usage. We reinforce the results from the Reuse Distance experiments in the previous Chapter with real cache data collected during runtime by monitoring the cache-refer and cache-miss measures. By analyzing both measures we can conclude that mini-batching is capable of reducing L1 and L2 misses in most cases.

❏ Reinforcing the findings from Chapter 6, the introduction of the mini-batching technique (the smallest mini-batch size tested) provides the best relative increase in performance. However, increasing the mini-batch size over 50 instances has diminishing returns in the performance gains, which can be seen by the plateau behavior in many charts from Fig. 21 and the smaller reduction in the cache-refer measure in Table 11.

❏ We performed experiments in three different platforms, with different cache sizes and memory bandwidth. The similarity in the behavior on all three platforms indicates that our solution is platform independent.

❏ Reinforcing the findings from Chapter 5, the most complex algorithms (i.e., the ones using the biggest lambda and employing additional strategies) can grow their models faster, which means they generate bigger models while training from the same original data. With bigger models, comes bigger resource consumption in the sense of data structures size and number of operations performed per instance. In such cases, the reuse of data already loaded into the faster memories is bigger and provides even more benefit by using our solution. Ultimately, because of these characteristics, the most complex algorithms are the most scalable.

❏ In a similar fashion, dataset characteristics that increase model size and complexity (e.g., the number of possible values from nominal attributes in Airlines dataset) have a tendency to present the best scalability.

❏ By using a total of six algorithms in our experimental setup we can confidently state that the proposed method is generic and applicable to a whole class of algorithms (i.e., bagging ensembles for data streams), since all algorithms presented similar behavior.

In summary, the results presented in this Chapter demonstrate that implementations of ensembles based on multi-core parallelism combined with mini-batching yield significant performance improvements. This is supported by the improvements in speedup and the reduction of cache misses in the L1 and L2 cache. The solution is applicable to a whole class of algorithms and works on hardware with different characteristics. However, many aspects of our solution remain to be explored.

For instance, from our experiments we could not provide a one size fits all for the mini-batch size. Different algorithms and datasets require different mini-batch sizes. Variations in mini-batch size may produce alterations in predictive performance as well as different outcomes regarding the computational performance. Thus, an adaptive mini-batch size could be an interesting addition to the technique. Another factor that could be used on such adaptive strategy is the energy-efficiency of our solution. In the next Chapter, we present a study on the energy-efficiency of our solution with the goal of having a better understanding of the energy consumption behavior.

Chapter 8

Improving energy-efficiency for data stream ensembles

In 2018, Information and Communication Technology (ICT) accounted for 3% of the global energy consumption, expecting to grow nearly 9% per year. This surge of energy consumption is happening because of the fast emergence of numerous applications and new ICT devices (GUEGAN; ORGERIE, 2019). The influx of applications and devices generates a significant demand to transmit, store, and process ever-increasing volumes of data. This scenario makes energy consumption an issue in virtually all applications, since both, small devices and data centers, can benefit from a better energy-efficiency. The benefit for small devices comes in the form of longer battery duration, for example, while in data centers it translates to a reduction of a big portion of their costs.

Among the most used applications of today, Machine Learning (ML) has been growing rapidly in the past few years. Historically, ML research focused on extracting the best predictive performance without giving much care for computational resources usage and energy consumption. However, the environment created by the new applications brings new challenges. These challenges invite researchers to develop new strategies capable of dealing with dynamic environments.

This shift towards data stream learning to address such challenges, has provided new algorithms like the Hoeffding Tree (HT) that is capable of adhering to requirements such as single pass processing, response time, and constant memory usage (DOMINGOS; HULTEN, 2000). Nevertheless, little effort has been made to reduce the energy consumption of such algorithms. In fact, given the sheer volume of data manipulated nowadays, it is clear that energy efficiency should be a goal in ML research, and it is starting to gain importance in state-of-art research (GARCíA-MARTíN; BIFET; LAVESSON, 2021). Therefore, the proposed method was evaluated regarding energy efficiency in a real data stream scenario. This section describes all the processes and characteristics of the energy-efficiency experiments. It is divided into four subsections: section 8.1 describes the hardware characteristics, the environment setup, and mentions the datasets and the software frameworks we have used. The section 8.2 presents the measures used to evaluate our experiments. Subsection 8.3 shows preliminary results used to calibrate and define a few parameters for the final experiments. Finally, the main findings regarding energy-efficiency are presented in section 8.4.

8.1 Environment

In this experiment, the initial focus was on constrained devices such as the Raspberry Pi 3 Model B. Later on, two more machines were added in order to test the method in a wider variety of scenarios. The new machines are typical representations of Data center and commodity hardware, indicating that the method is generic and provides benefits in many different contexts. Hardware specifications of the aforementioned machines are shown in Table 12.

Table 12 – Hardware specifications

Processor	Xeon 4208	i5-2400	Cortex-A53
Cores/socket	8	4	4
Clock frequency (GHz)	2.1	3.1	1.2
L1 cache (core)	32 KB	128 KB	32 KB
L2 cache (core)	1024 KB	1024 KB	512 KB
L3 cache (shared)	11264 KB	6144 KB	-
Memory (GB)	128	4	1
Memory channels	6	2	-
Maximum bandwidth	107.3 GiB/s	21 GB/s	-

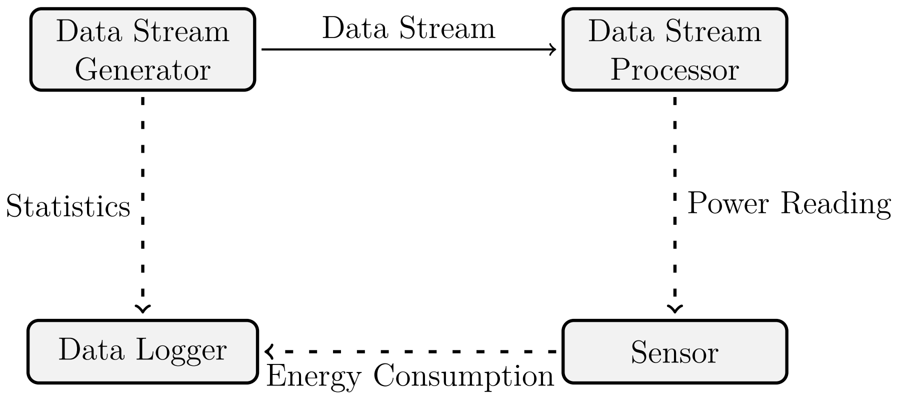


Figure 22 – Logic diagram of the experiment environment.

The experimental environment consists of an isolated network connected through a dedicated switch. The readings are based on a Yokogawa MW-100, a scalable, high-performance data acquisition platform. The sensors collect instant W consumption from power plugs and make this information accessible through the network. There are three sensors providing readings from the three machines described in Table 12. Each machine from Table 12 is an instance of a *Data Stream Processor* node on Fig. 22. Also, there is a machine responsible for collecting W consumption data from the sensors. It accomplishes this task by querying the sensor, appending this information to a file, and sleeping until the next reading.

Another machine (*Data Stream Generator*) is responsible for creating a parameterizable load and sending it through sockets to the Data Stream Processors. The operation summary of the Data Stream Generator is as follows: it reads the whole dataset from the disk and stores it in memory, then opens a connection with the Data Stream Processor node and starts sending data asynchronously at a parameterizable rate received as an argument. The generator sends data until the end of the dataset or until it reaches the time limit. Upon reaching the termination point, the generator sends a termination message and stores statistics like the total number of instances sent. The generator produces instances at a rate defined by a parameter called Instances Per Second (IPS). The generator sends data in five smaller batches at constant intervals (e.g., a rate of 100 IPS will send 20 instances each 200ms to the consumer). Lastly, the *Data Logger* aggregates data from every node.

Once again, the technique is implemented in the Massive Online Analysis (MOA) framework (BIFET et al., 2010), and tested on six bagging ensemble algorithms (described in section 2.2) with five datasets (described in section **??**). We chose MOA[1] framework as it provides several ensemble learners for data stream processing, with the added benefit of having been used for many studies in the ML area (BIFET et al., 2010). This choice allows us to assess the efficiency of the *mini-batching* strategy for several ensemble algorithms.

8.2 Evaluation measures

This subsection defines the five measures used to evaluate the performance of the technique in the experiments. The measures are: Joules Per Instance (JPI), average Watt (W) consumption, Instances Per Second (IPS), accuracy, and average Delay to process.

JPI is one of the measures related to energy consumption. However, Before defining JPI it is necessary to define energy consumption. In the context of computer systems, energy is delivered as electricity. However, most energy consumption monitors operate by collecting an instantaneous rate of Power (W) being supplied. Energy, on the other hand, is expressed in Joules (J). Its value is the product of power (W) and time (t), as shown in the following equation:

[1] Available at https://github.com/Waikato/moa

$$J = W \times t$$

Since the power supply can fluctuate, it is necessary to get frequent readings at small enough intervals to correctly calculate the consumption. Thus, the total energy consumption can be approximated to:

$$E = \sum_{i=1}^{n} P_i \times t,$$

where n is the number of samples taken by the monitor.

Typically, *energy efficiency* is defined as a relation between the computing work done and the amount of energy spent. Given the amount of varying parameters we have used, we decided to measure computing work by the number of instances processed within a time interval. Therefore, we define the JPI measure as:

$$JPI = \frac{E}{n},$$

where n is the number of instances processed and E is the amount of energy spent. Since this measure reflects the amount of energy consumed to process each instance, the smaller this measure is the better the energy efficiency.

The second measure related to energy efficiency is the Average Watt consumption, defined as the average of all the readings made by the monitor during a given experiment. Its formula can be defined as:

$$AWC = \frac{\sum_{i=1}^{n} W}{n},$$

where W represents each reading and n represents the total number of readings.

IPS is a measure related to throughput, defined as the number of instances processed per second. It is defined as:

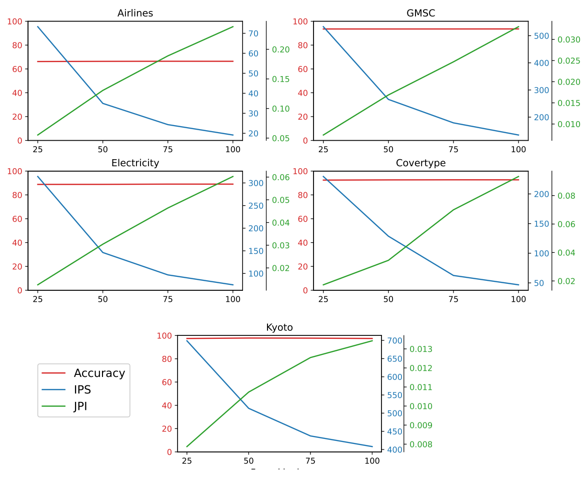
$$IPS = \frac{n}{s},$$

where n is the number of instances processed and s the total time taken in seconds.

The fourth measure used is the average delay, which is the average time to process each instance from when it arrives through the socket until its training is processed. Average Delay is used to show the impact caused by the mini-batching approach in the response time. Is can be defined as:

$$Avg\ Delay = \frac{\sum_{i=1}^{n} ILT}{n},$$

where ILT is the Instance Living Time, calculated as $ILT = FPT - AT$, where FPT is the Finish Processing Time of the instance and AT is the Arrival Time of the instance.


(a) Using ARF algorithm

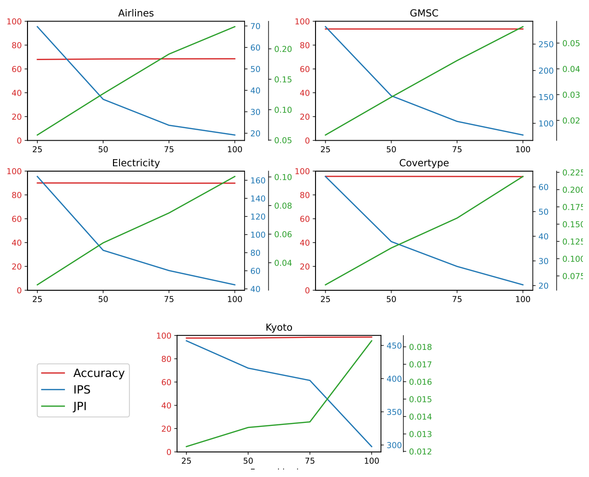

(b) Using SRP algorithm

Figure 23 – Accuracy, energy consumption and throughput according to ensemble size for each dataset on Raspberry Pi model 3 B.

The last measure to be presented is accuracy, a well known measure that informs the amount of correct classifications. It is defined as:

$$Accuracy = \frac{correctly\ classified}{n},$$

where n is the number of instances processed.

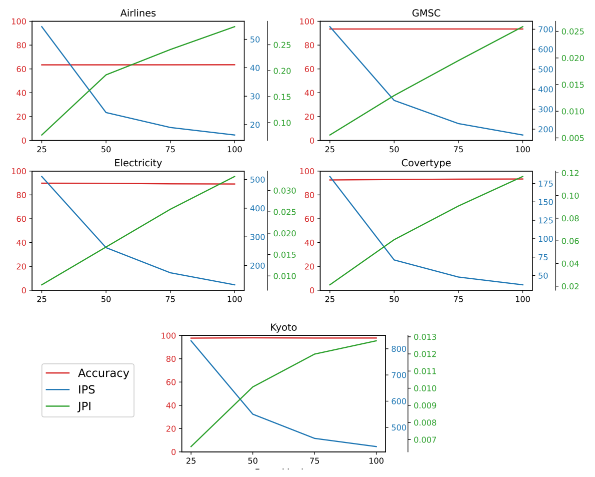
8.3 Preliminary results

Two preliminary experiments were made, the first to find a good ensemble size to operate with in constrained devices and the second to understand the energy consumption profile of all hardware configurations.

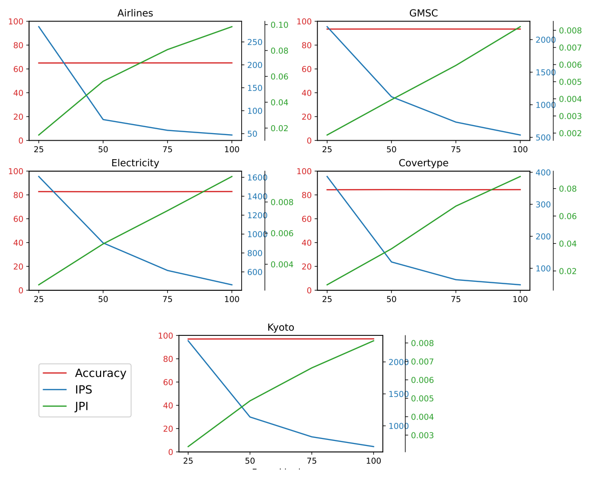
The goal of the first experiment was to find a good trade-off involving accuracy, energy consumption, and throughput according to the ensemble size when using a constrained device. In this experiment, the baseline version of all the algorithms was executed using all datasets while measuring energy consumption and calculating throughput using the IPS measure. The results are shown in Figures 23, 24, and 25, where the X-axis shows the ensemble size and the Y-axis has three different scales as shown in the legend.

It is possible to summarize the behaviors shown in Figs. 23, 24, and 25 with three trends, all three of them related to the increase of the ensemble size (X-axis). Accuracy (in red) decreases by a very small margin, remaining almost constant. Throughput (in blue) decreases and energy consumption (in green) increases. Based on these behaviors, it is acceptable to use smaller ensembles on constrained devices where energy and computational power are limited resources.

The second experiment's goal was to determine the energy consumption profile of each machine. The energy consumption was monitored while varying the resource us-

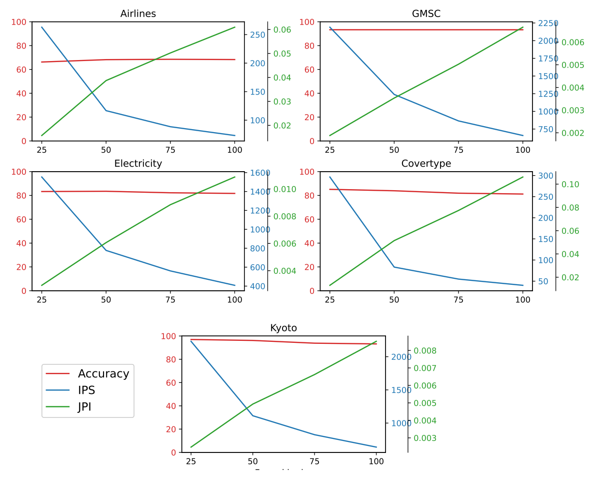


(a) Using LBag algorithm

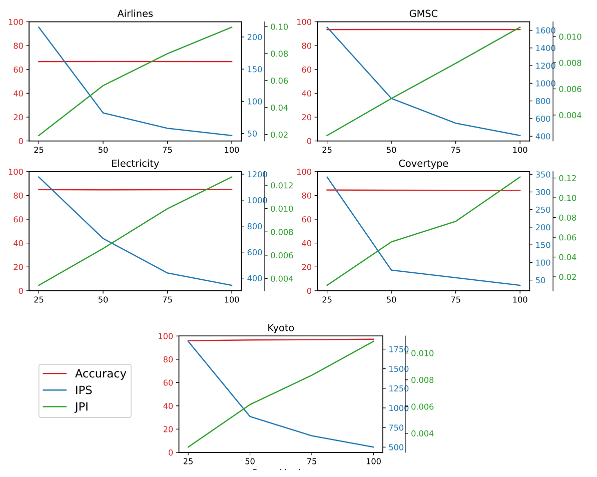


(b) Using OB algorithm

Figure 24 – Accuracy, energy consumption and throughput according to ensemble size for each dataset on Raspberry Pi model 3 B.



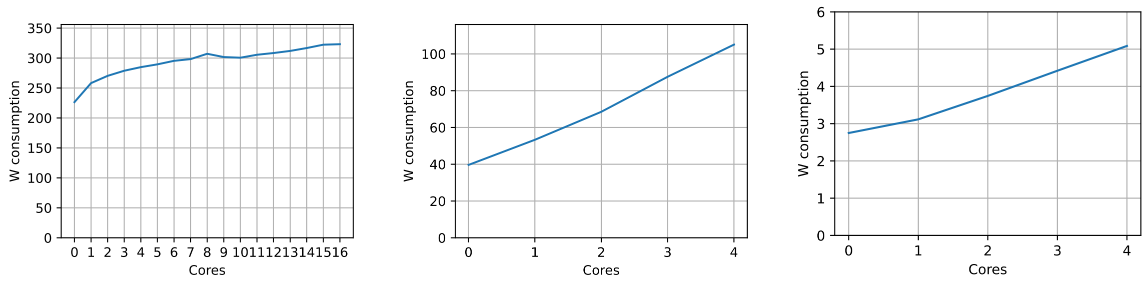
(a) Using OBASHT algorithm



(b) Using OBAdwin algorithm

Figure 25 – Accuracy, energy consumption and throughput according to ensemble size for each dataset on Raspberry Pi model 3 B.

age with the aid of the Stress tool for Linux. The resource usage started from idle and was increased until the maximum number of cores available on the machine were being used. After measuring each workload for 180 seconds, the average W consumption was calculated. The results can be seen in Figure 26, where it is possible to confirm some expected aspects, like the Raspberry Pi showing a much smaller consumption than the other machines. We found that our measurements from the Xeon Silver 4208 machine suffers significant interference from peripherals since it has many features embedded. In the chip specification, the Thermal Design Power (TDP) is 80, however, the machine is a server that has two chips with a lot of fans and other auxiliary systems that consume power. The biggest takeaway from this experiment is that, generally, the closer to 100% utilization, the better the energy efficiency.



(a) Xeon Silver 4208 (b) i5-2400 (c) Pi 3 model B

Figure 26 – Energy consumption profile for each machine.

8.4 Energy efficiency in streaming ensembles

To better assess the energy consumption behavior of the three machines, a data stream was simulated under three different levels of load (e.g., receiving instances through the network with different rates) when applying the six algorithms on five datasets. This experiment used the setup described in subsection 8.1 and a modified MOA version that receives instances through sockets instead of reading from an ARFF file.

The bagging ensemble algorithms have different characteristics among themselves, as explained in Section 2.2, leading to different throughput levels. Also, the machines used have vastly different processing power, and even the datasets generate models with varying levels of both memory and CPU requirements. In this scenario, the generator has to produce different workloads for each case, according to the combination of machine, algorithm, and dataset in use. The throughput when reading from the file was considered to be the maximum. Then the three rates (e.g., 90%, 50% and 10%) were applied for each combination of machine, algorithm and dataset to calibrate the different workloads.

Since this approach resulted in highly variable generator rates, we opted to show the results using the JPI and delay measures.

In these experiments, we compare three versions, the baseline(Sequential), a parallel version without mini-batch (B1), and a parallel version with a mini-batch of 500 instances (B500). Figures 27, 28 and 29 present the results from Raspberry Pi, i5, and Xeon Silver 4208, respectively. In these figures, each line shows charts of a specific dataset and each column is associated with a specific algorithm (e.g., the first line has all charts from the experiments with the airlines dataset, whereas the first column has all experiments with ARF). All charts in the same line have the same scale on both Y-axis. The left Y-axis displays the JPI's scale, while the right Y-axis displays the Delay's scale.

As a general remark, energy efficiency is closely related to model complexity. Which, as mentioned before, is influenced by the algorithm and dataset. To allow an easier comparison between algorithms, all charts in a line have the same scale and were plotted from experiments with the same dataset. It is possible to see that all three versions of OzaBag present a better energy efficiency (i.e., smaller JPI) compared to the other three

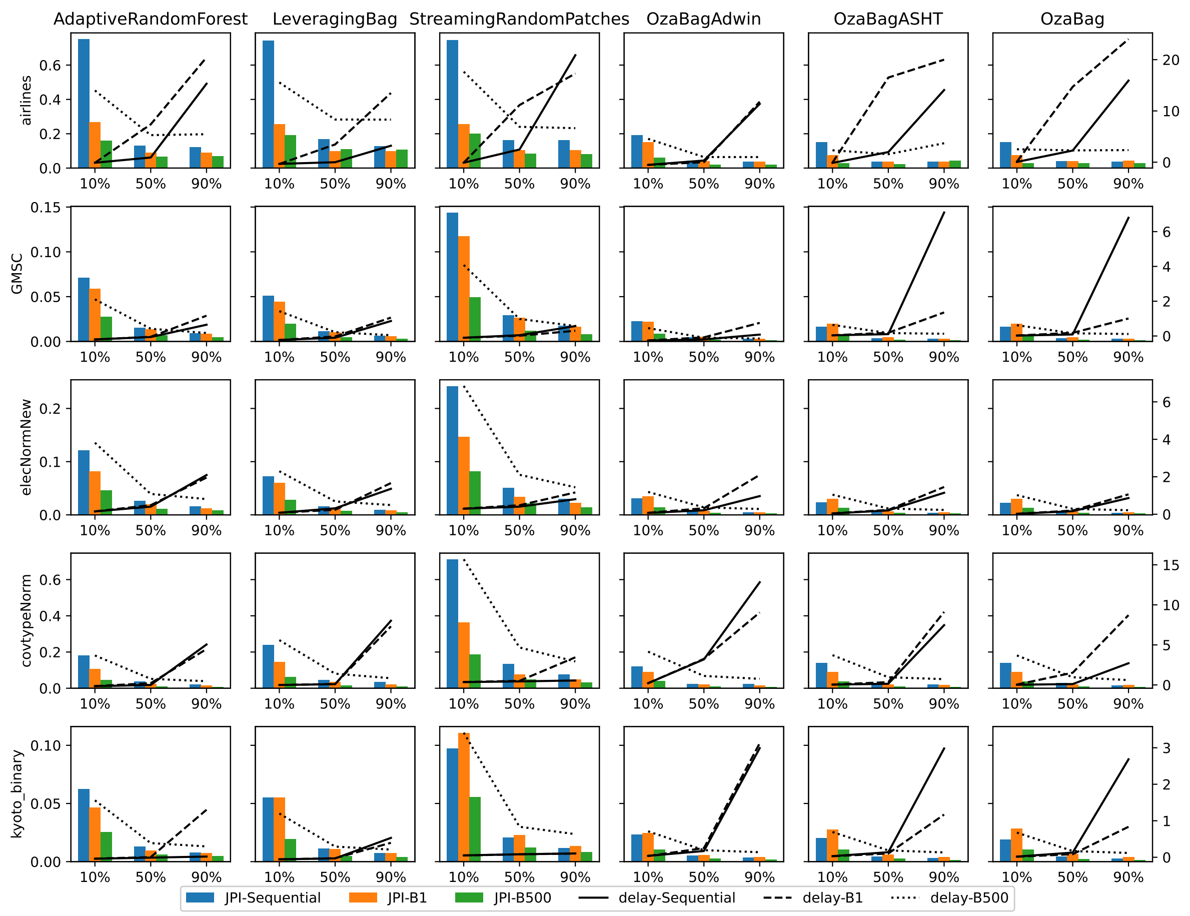


Figure 27 – Energy consumption and delay for the Raspberry Pi

algorithms. The better energy efficiency presented by OzaBag variants is caused by their smaller models that require fewer memory operations and allows a faster traversal of the trees.

On the other hand, more complex algorithms tend to present a bigger percentage reduction in the JPI measure from the best non-mini-batch version to the mini-batch version. This behavior is related to the higher throughput produced by the mini-batch version, which shortens the execution time. The mini-batching strategy creates more sleep periods at lower rates while waiting for more instances to reach the mini-batches desired size. When the rate of incoming instances is several times smaller than the mini-batch size, it may lead to higher delays. Although mini-batching can improve energy efficiency, the delay resulting from extended and repeated sleep periods may hinder the idea of real-time processing.

Another easily noticeable trend is that SRP has the worst energy efficiency (i.e., highest JPI) across all the experiments, independently of the platform and algorithm used. This effect occurs because this algorithm tends to create deeper trees in less time than the other complex algorithms (e.g., ARF and LBag), leading to an increase in computational complexity as a whole.

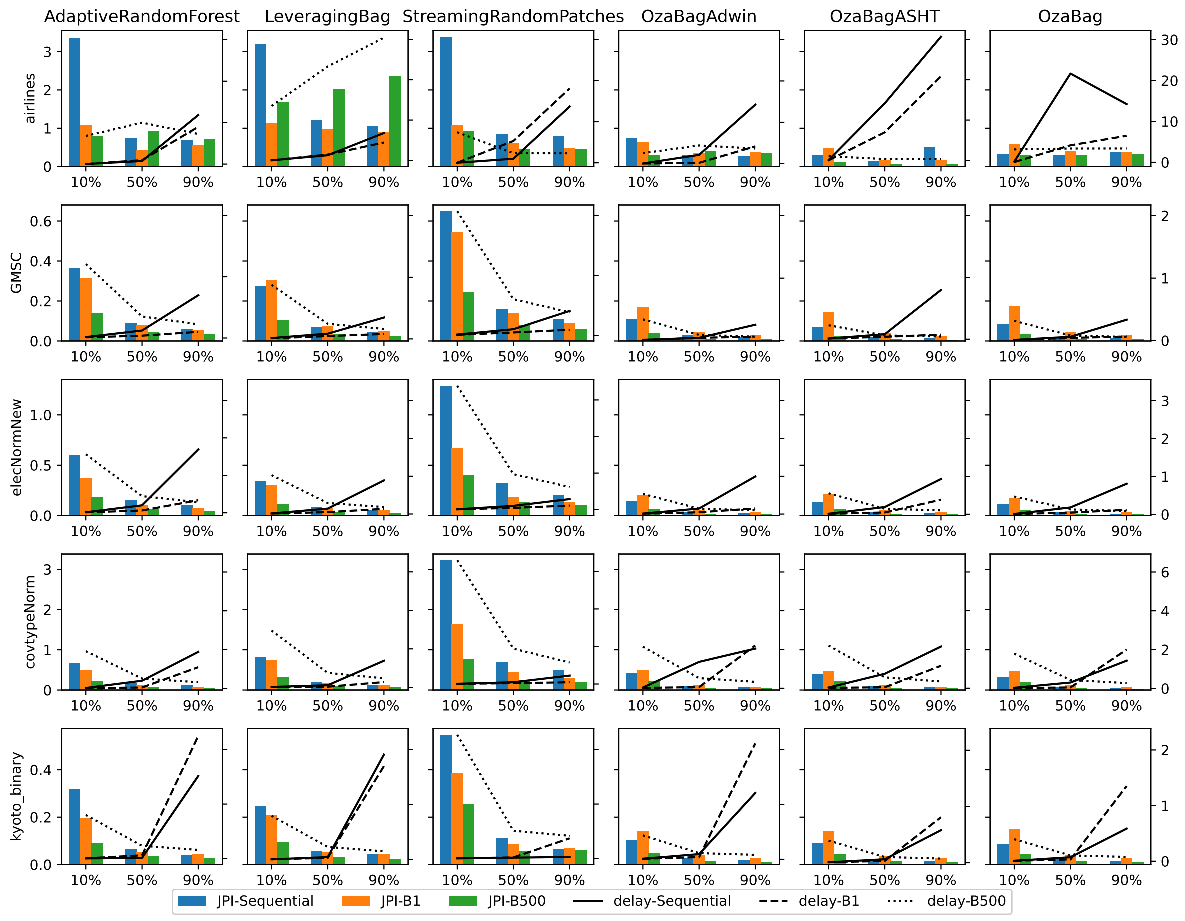


Figure 28 – Energy consumption and delay for the Vostro

In addition to the charts, Tables 15, 14 and 13 present the percentage Δ in JPI between the best version without mini-batch (i.e., the best case between the orange and blue bars from the charts) and the mini-batch version. A negative value indicates the mini-batch version reduced energy consumption by that percentage amount. Cases where mini-batch consumed more energy have positive values and are in bold.

Based on the data presented in the tables, It is possible to say that using mini-batch reduces energy consumption in the vast majority of cases. More often than not, mini-batching presents a more significant reduction in energy consumption at lower incoming rates. As stated before, the three more complex algorithms (e.g., ARF, LBag, and SRP) have a slightly more significant percentage reduction than the OzaBag versions. This behavior happens because their more complex models benefit the most from the improved memory locality provided by the mini-batching technique.

In summary, our mini-batching proposal has shown improvements in energy consumption across all experiments. While the average delay for the mini-batching algorithm tends to drop as the rate increases, both sequential and parallel without mini-batch algorithms tend to present an increase when the rate gets past 50%. The increase in non-mini-batching versions happens because instead of receiving one instance, the algorithm receives a group

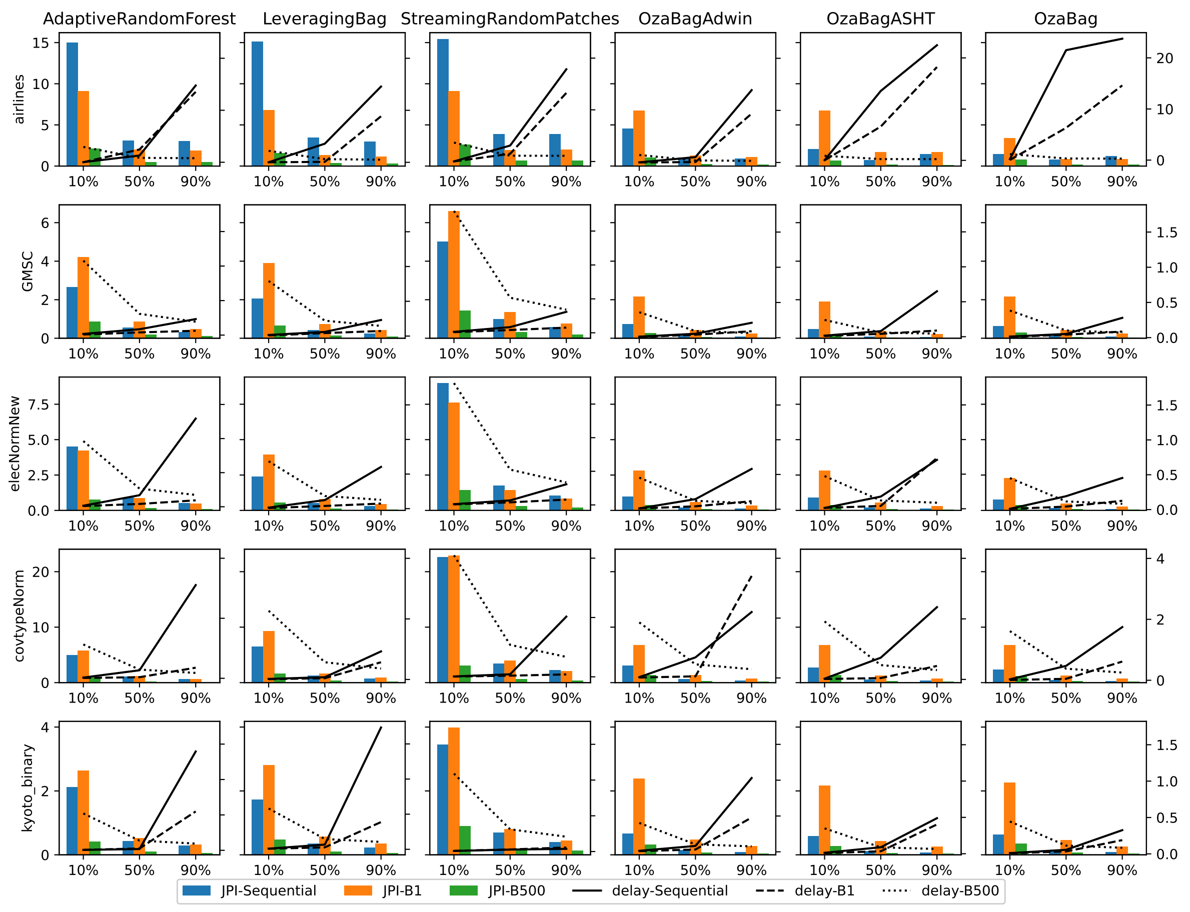


Figure 29 – Energy consumption and delay for the Xeon

of instances, causing the last instances of the mini-batch to have a longer living time inside the system.

Algorithm	Rate	Airlines	GMSC	Electricity	Covertype	Kyoto
Ada	10	-105.89	-31.54	-35.77	-61.53	-20.81
	50	-22.12	-7.15	-8.23	-13.83	-3.41
	90	-18.75	-4.09	-3.90	-7.92	-2.35
L	10	-63.62	-24.50	-32.12	-82.12	-35.62
	50	**11.53**	-5.21	-6.00	-18.84	-6.16
	90	**10.56**	-2.93	-3.19	-12.65	-3.53
Patches	10	-55.26	-67.94	-64.77	-177.65	-41.81
	50	-18.49	-14.95	-12.94	-29.27	-8.70
	90	-22.79	-8.48	-7.84	-17.97	-3.30
Adwin	10	-90.51	-13.47	-16.79	-50.12	-12.73
	50	-19.98	-2.86	-3.55	-12.42	-2.79
	90	-17.63	-1.51	-1.89	-10.64	-1.87
ASHT	10	-46.35	-9.14	-10.63	-50.65	-9.53
	50	-14.32	-1.85	-2.19	-11.62	-2.07
	90	**6.42**	-1.72	-1.12	-10.36	-1.52
OzaBag	10	-45.44	-9.10	-10.17	-52.24	-8.77
	50	-10.67	-1.86	-2.07	-11.61	-1.88
	90	-7.69	-1.66	-1.14	-10.50	-1.30

Table 13 – Percentage difference between the best non-mini-batch version and the mini-batch version on the Raspberry Pi

Algorithm	Rate	Airlines	GMSC	Electricity	Covertype	Kyoto
Ada	10	-0.29	-0.17	-0.18	-0.27	-0.11
	50	**0.49**	-0.04	-0.04	-0.06	-0.02
	90	**0.16**	-0.02	-0.02	-0.04	-0.02
L	10	**0.54**	-0.17	-0.18	-0.41	-0.12
	50	**1.04**	-0.04	-0.04	-0.09	-0.02
	90	**1.48**	-0.02	-0.03	-0.05	-0.02
Patches	10	-0.16	-0.30	-0.27	-0.86	-0.13
	50	-0.15	-0.06	-0.05	-0.19	-0.03
	90	-0.04	-0.03	-0.03	-0.11	-0.00
Adwin	10	-0.35	-0.07	-0.09	-0.19	-0.05
	50	**0.10**	-0.02	-0.02	-0.04	-0.01
	90	**0.09**	-0.01	-0.01	-0.02	-0.01
ASHT	10	-0.19	-0.05	-0.07	-0.16	-0.05
	50	-0.08	-0.01	-0.02	-0.04	-0.01
	90	-0.11	-0.01	-0.01	-0.02	-0.01
OzaBag	10	-0.02	-0.05	-0.06	-0.14	-0.04
	50	**0.02**	-0.01	-0.01	-0.03	-0.01
	90	-0.05	-0.01	-0.01	-0.02	-0.00

Table 14 – Percentage difference between the best non-mini-batch version and the mini-batch version on the Vostro

Algorithm	Rate	Airlines	GMSC	Electricity	Covertype	Kyoto
Ada	10	-6.97	-1.78	-3.44	-4.12	-1.70
	50	-1.49	-0.38	-0.70	-0.91	-0.33
	90	-1.38	-0.22	-0.39	-0.51	-0.23
L	10	-5.20	-1.40	-1.82	-4.81	-1.26
	50	-0.96	-0.29	-0.38	-0.91	-0.25
	90	-0.85	-0.18	-0.23	-0.52	-0.16
Patches	10	-6.53	-3.58	-6.17	-19.51	-2.54
	50	-1.29	-0.68	-1.13	-2.78	-0.50
	90	-1.33	-0.39	-0.63	-1.73	-0.27
Adwin	10	-3.54	-0.47	-0.63	-1.75	-0.34
	50	-0.78	-0.10	-0.13	-0.36	-0.08
	90	-0.72	-0.06	-0.08	-0.21	-0.04
ASHT	10	-1.43	-0.29	-0.53	-1.31	-0.31
	50	-0.56	-0.06	-0.11	-0.29	-0.07
	90	-1.33	-0.04	-0.07	-0.16	-0.04
OzaBag	10	-0.63	-0.32	-0.41	-1.16	-0.28
	50	-0.58	-0.07	-0.09	-0.23	-0.06
	90	-0.66	-0.04	-0.06	-0.14	-0.03

Table 15 – Percentage difference between the best non-mini-batch version and the mini-batch version on the Xeon

Chapter 9

Conclusion

Ensemble learning is a fruitful approach to improve the performance of ML models by combining several single models. Although ensembles are popular for yielding highly accurate results, many aspects of their efficient implementation remain to be studied. In this work, we proposed a task-parallel model coupled with the *mini-batching* technique for improving online bagging ensembles.

Although bagging ensembles have independent models that provide very natural parallelism, this solution alone is insufficient to achieve good results. In Chapter 3, we present several results where the speedup achieved is small or even nonexistent. Such results reject our **H1** which stated that a task-parallel model could improve execution time.

When coupled with a mini-batching technique capable of improving the memory access patterns, it is possible to reduce the execution time and, as a bonus, increase the energy efficiency of such algorithms. We demonstrated that the performance achieved by multi-core parallelism could be remarkably improved by applying the mini-batching technique using theoretical and experimental frameworks. We used different hardware platforms as often as possible in our experimental frameworks. We believe the wide range of hardware platforms, algorithms, and datasets used in the experimental frameworks, as well as the many different scenarios tested and auxiliary results presented (e.g., real data-stream under different loads, standard benchmarks with different sizes, impact on predictive performance, impact on change detections, energy-efficiency, etc.), provide sufficient evidence that the proposed method applies to the whole class of bagging ensembles and can be deployed in many different environments. Thus, we confirm our **H2** which stated that better memory access patterns lead to better computational performance.

We also presented a study regarding the trade-off when adopting the mini-batching technique. Using mini-batching in data stream bagging ensembles has an impact on their predictive performance. However, the intensity of such impacts varies in tandem with other experiment parameters, with the characteristics from the dataset being the most critical variable. Nevertheless, small mini-batches (e.g., up to 50 examples) are a safe bet to improve execution performance by at least 2 to 3 times while losing less than 1% predictive performance. Such results confirm our **H3** which stated that it is possible to accelerate bagging ensembles for data streams without significantly impacting their predictive performance.

We demonstrated that energy consumption could be improved by the proposed method. We used different workloads to assess the energy efficiency at various stress levels. Regarding the amount of energy economized, the best result happens on low workloads at the cost of a higher delay in the response time. On the other hand, higher workloads tend to make the energy consumption levels more similar between the default and the proposed method, usually presenting a smaller delay in favor of our method.

9.1 Future work

As much as we tried to cover a wide range of variables, there are improvements to be made on the method and behaviors that could receive further investigation to achieve complete understanding. We present some of these ideas for future work in this Section.

Temporary data structures management: One of the major contributions of this thesis is the adoption of the mini-batching technique to improve the memory efficiency of the bagging ensembles. We mitigated the memory bottleneck created by sequentially loading the classifiers into the higher memory hierarchies to train a single instance. However, other aspects of memory usage could be improved, such as the management of temporary objects created during the training step. To achieve this, we need an auxiliary control system for the temporary structures. Such a system would store the discarded structures in a collection instead of letting the garbage collector (GC) destroy them. It would also be in charge of taking an unused structure from the collection and updating with the new data when requested. If the collection has no discarded structure left (i.e., is empty), then, and only then, a new structure would be created through the normal JVM means. Typically, the GC is optimized enough that we do not need to worry about its work. However, the sheer volume of objects created and freed per second (as shown in Figure 15) in these algorithms may create the circumstances in which a more fine-grained control is needed. Using this middleware system for data structure management, we could alleviate the pressure on GC with the added benefit of less memory fragmentation. In addition, it could be possible to implement it in a way that such collection of discarded data structures could be (pre)allocated in contiguous memory, which increases the speed of memory operations.

Algorithms with efficient data structures: Another possible future work is to devise and implement algorithms that use very efficient data structures for the models. Such data structures should be small enough to “live” in the cache memory during the whole time. This solution could eliminate the bottleneck of transferring data from the main memory to the higher hierarchies. The challenge in this case is to ensure the quality of the information stored using such a small amount of memory.

Parallelization with mini-batching of the classification step: In this thesis, we approached the problem of how to implement parallelism in the training step of bagging ensembles efficiently. However, due to time and scope limitations, we could not include experiments with parallel classification in our evaluation framework. To justify the parallelization of the classification step, one would have to change the underlying classifiers from decision trees to models that consume more resources during classification when they need to output the predictions. Although not predominant, such methods are commonly used. The parallelization of classification would also be straightforward because each classifier is still independent in bagging ensembles. However, the classification step has the aggregation process, which serves as a natural synchronization for the parallel portion. Yet, the aggregation provides an opportunity for further optimization in the form of a *Reduce* step on the partial results from the terminated prediction tasks. This extra layer of complexity (i.e., having the prediction tasks and the Reduce task) could provide even more performance gains, as the final predictions could be computed partially instead of waiting for all parallel tasks to finish. Nevertheless, one would need to perform experiments with and without the Reduce step during aggregation to claim that such a strategy provides benefits.

Utilization of optimized classifiers as the ensemble base models: In theory, one could use an existing optimized method for a specific classifier in conjunction with our solutions (as mentioned in the Chapter 4). That is, as long as the optimized/improved classifier is the base model of the bagging ensemble. It would be interesting to implement one optimization solution coupled with our parallel mini-batching technique and perform experiments in the same evaluation framework we used. Such experiments would allow the identification of relationships among methods. For instance, in exceptional cases, one could find superlinear speedups (as is the case of LeveragingBag classifying the Airlines dataset). Even more, the possibility of choosing different optimization techniques for the base models from a repository would be very attractive for exploratory tasks. Although not as straightforward to implement, one could even combine ensemble optimizations with our solutions. In essence, this venue for future work is more focused on the combination of existing techniques.

Adaptive mini-batch size: The mini-batch size is a critical parameter in our solution. However, we could not provide a clear guide on properly tuning it because the optimization can consider many aspects. For instance, if the goal is the best predictive performance, a mini-batch of 10 instances might be the best option since the difference to the baseline will be minimal. On the other hand, the largest mini-batch size would achieve the best energy efficiency and the fastest execution time at the expense of latency and prediction performance. In addition, we can safely say that the ideal mini-batch size will be different according to variables that can not be controlled, such as the volatility of the dataset, how often drift occurs, and the dataset composition regarding the attribute types. Therefore, the mini-batch size tuning is an example of multi-objective optimization (MOP), where we have to optimize multiple objective functions simultaneously.

Since we are working in the data stream context, we do not know many variables before runtime, unlike the static context, where we could optimize the mini-batch size by knowing all variables. This difference increases the difficulty and complexity of finding the optimal mini-batch size. Given the volatile characteristics of data streams, it is safe to assume that, even if the goal remains the same, the optimal mini-batch size can change as the data distribution changes. Therefore, we need to exclude the outdated data to ensure that the optimal mini-batch size is calculated using only the current data distribution. One way to implement such behavior is by adopting sliding windows such as the ones used in ADWIN change detection. Alternatively, we could discard the historical data every time a change is detected in data distribution.

One of the basic objectives we have when setting the mini-batch size is maintaining the quality of the predictions. Based on the experiments performed for this thesis, evidence indicates that some data characteristics will have a negligible impact on the predictive performance when we increase the mini-batch size, whereas data with slightly different characteristics may show significant degradation in the predictive performance. Once again, the volatile nature of data streams creates the need to use sliding windows while monitoring the predictive performance. In this way, if the predictive performance stays the same over a few windows, it is safe to assume that we can increase the mini-batch size by some amount without the fear of degrading it. The opposite is also true. If the predictive performance starts to degrade, then we have to decrease the size of the mini-batch.

The execution time can also be considered an objective function. In contrary to the predictive performance objective, we need to increase the mini-batch size to improve the outcome of this objective function. However, we need to monitor several context data from our application (e.g., input rate, throughput, and delay to process each instance) to optimize the mini-batch size regarding the execution time. While monitoring only one aspect (the prediction quality) of the application was sufficient to correctly estimate the action to optimize the mini-batch size regarding the predictive performance, For example, if both the difference between throughput and input rate and the delay to process are increasing, we have to increase the mini-batch size to process faster and catch up to the stream. Alternatively, if we have a minimal delay and the throughput matches the input rate, we can try to reduce the mini-batch size to improve the prediction quality.

Lastly, the energy consumption could also be an objective function, and its behavior would be similar to the execution time. The context data of interest, in this case, is the energy consumption of the current window, and we should increase the mini-batch size to save energy.

In essence, the adaptive behavior would be based on incremental steps based on heuristics applied to the context data collected in the current window. A more advanced option would be to give the user an option to choose the objective functions of the adaptive behavior. For example, it would allow the user to disable the energy consumption objective function, so optimizing the mini-batch size would only consider the predictive performance and execution time. In addition, the user could define weights for each objective function allowing the creation of a priority list/system in the objective functions. In summary, such a system would only make sense if deployed over a long time.

Bibliography

AGGARWAL, C. C. **Data streams - models and algorithms**. [S.l.]: Springer US, 2007.

AL-KHATEEB, T. et al. Stream classification with recurring and novel class detection using class-based ensemble. In: **2012 IEEE 12th International Conference on Data Mining**. [S.l.: s.n.], 2012. p. 31–40.

AMEZZANE, I. et al. Energy consumption of batch and online data stream learning models for smartphone-based human activity recognition. In: **2019 4th World Conference on Complex Systems (WCCS)**. [S.l.: s.n.], 2019. p. 1–5.

BABCOCK, B. et al. Models and issues in data stream systems. In: **Proceedings of the Twenty-First ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems**. New York, NY, USA: Association for Computing Machinery, 2002. (PODS '02), p. 1–16. ISBN 1581135076. Disponível em: <https://doi.org/10.1145/543613.543615>.

BACKUS, J. The history of fortran i, ii, and iii. In: _____. **History of Programming Languages**. New York, NY, USA: Association for Computing Machinery, 1978. p. 25–74. ISBN 0127450408. Disponível em: <https://doi.org/10.1145/800025.1198345>.

BALAKRISHNAN, G.; SOLIHIN, Y. West: Cloning data cache behavior using stochastic traces. In: IEEE. **IEEE International Symposium on High-Performance Comp Architecture**. [S.l.], 2012. p. 1–12.

BARDDAL, J. P. et al. A survey on feature drift adaptation: Definition, benchmark, challenges and future directions. **Journal of Systems and Software**, v. 127, 07 2016.

BASILICO, J. D. et al. Comet: A recipe for learning and using large ensembles on massive data. In: **2011 IEEE 11th International Conference on Data Mining**. [S.l.: s.n.], 2011. p. 41–50.

BEN-HAIM, Y.; TOM-TOV, E. A streaming parallel decision tree algorithm. **Journal of Machine Learning Research**, v. 11, n. 28, p. 849–872, 2010. Disponível em: <http://jmlr.org/papers/v11/ben-haim10a.html>.

BIFET, A.; GAVALDà, R. Learning from time-changing data with adaptive windowing. In: **Proceedings of the 7th SIAM International Conference on Data Mining**. [S.l.: s.n.], 2007. v. 7.

BIFET, A.; HOLMES, G.; PFAHRINGER, B. Leveraging bagging for evolving data streams. In: BALCÁZAR, J. L. et al. (Ed.). **Machine Learning and Knowledge Discovery in Databases**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. p. 135–150. ISBN 978-3-642-15880-3.

BIFET, A. et al. New ensemble methods for evolving data streams. In: **Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining**. [S.l.: s.n.], 2009. p. 139–148.

_____. Moa: Massive online analysis, a framework for stream classification and clustering. In: **Proceedings of the First Workshop on Applications of Pattern Analysis**. [S.l.: s.n.], 2010. p. 44–50.

BONACIC, C. et al. Multithreaded processing in dynamic inverted indexes for web search engines. In: ACM. **Proceedings of the 2015 Workshop on Large-Scale and Distributed System for Information Retrieval**. [S.l.], 2015. (Proceedings of the 2015 Workshop on Large-Scale and Distributed System for Information Retrieval), p. 15–20.

BREIMAN, L. Bagging predictors. **Machine Learning**, v. 24, n. 2, p. 123–140, 1996. ISSN 1573-0565. Disponível em: <https://doi.org/10.1007/BF00058655>.

_____. Random forests. **Machine learning**, Springer, v. 45, n. 1, p. 5–32, 2001.

BUTENHOF, D. R. **Programming with POSIX threads**. [S.l.]: Addison-Wesley Professional, 1997.

CARBONE, P. et al. Apache flink: Stream and batch processing in a single engine. **Bulletin of the IEEE Computer Society Technical Committee on Data Engineering**, IEEE Computer Society, v. 36, n. 4, 2015.

CASSALES, G. et al. Improving parallel performance of ensemble learners for streaming data through data locality with mini-batching. In: **IEEE International Conference on High Performance Computing and Communications (HPCC)**. [S.l.: s.n.], 2020.

_____. Improving the performance of bagging ensembles for data streams through mini-batching. **Information Sciences**, v. 580, p. 260–282, 2021. ISSN 0020-0255. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0020025521008938>.

CASSALES, G. W. et al. Idsa-iot: An intrusion detection system architecture for iot networks. In: **2019 IEEE Symposium on Computers and Communications (ISCC)**. [S.l.: s.n.], 2019. p. 1–7.

CHAN, P. K.; STOLFO, S. J. Toward parallel and distributed learning by meta-learning. In: . [S.l.]: AAAI Press, 1993. (AAAIWS'93), p. 227–240.

CHANDRA, R. et al. **Parallel programming in OpenMP**. [S.l.]: Morgan kaufmann, 2001.

CHEN, L.; ZOU, L.-J.; TU, L. Stream data classification using improved fisher discriminate analysis. **Journal of Computers**, 01 2009.

CYGANEK, B.; SOCHA, K. Novel parallel algorithm for object recognition with the ensemble of classifiers based on the higher-order singular value decomposition of prototype pattern tensors. In: **2014 International Conference on Computer Vision Theory and Applications (VISAPP)**. [S.l.: s.n.], 2014. v. 2, p. 648–653.

DENNING, P. J. The working set model for program behavior. **Commun. ACM**, Association for Computing Machinery, New York, NY, USA, v. 11, n. 5, p. 323–333, maio 1968. ISSN 0001-0782. Disponível em: <https://doi.org/10.1145/363095.363141>.

DENNING, P. J.; MARTELL, C. H. **Great principles of computing**. [S.l.]: MIT Press, 2015.

DENNING, P. J.; SLUTZ, D. R. Generalized working sets for segment reference strings. **Communications of the ACM**, ACM New York, NY, USA, v. 21, n. 9, p. 750–759, 1978.

DOMINGOS, P.; HULTEN, G. Mining high-speed data streams. In: ACM SIGKDD. **Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining**. [S.l.], 2000. p. 71–80.

DUNCAN, R. A survey of parallel computer architectures. **Computer**, v. 23, n. 2, p. 5–16, 1990.

EKANAYAKE, S. et al. Java thread and process performance for parallel machine learning on multicore hpc clusters. In: **2016 IEEE International Conference on Big Data (Big Data)**. [S.l.: s.n.], 2016. p. 347–354. ISSN null.

FARID, D. M. et al. An adaptive ensemble classifier for mining concept drifting data streams. **Expert Syst. Appl.**, Pergamon Press, Inc., USA, v. 40, n. 15, p. 5895–5906, nov. 2013. ISSN 0957-4174. Disponível em: <https://doi.org/10.1016/j.eswa.2013.05.001>.

FLYNN, M. J. Some computer organizations and their effectiveness. **IEEE Transactions on Computers**, C-21, n. 9, p. 948–960, 1972.

FORUM, M. P. I. Mpi: A message - passing interface standard. In: . [S.l.: s.n.], 1994.

FOSTER, I. **Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering**. USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN 0201575949.

FREUND, Y.; SCHAPIRE, R. E. A decision-theoretic generalization of on-line learning and an application to boosting. **Journal of Computer and System Sciences**, v. 55, n. 1, p. 119–139, 1997. ISSN 0022-0000. Disponível em: <https://www.sciencedirect.com/science/article/pii/S002200009791504X>.

FREUND, Y.; SCHAPIRE, R. E. et al. Experiments with a new boosting algorithm. In: **ICML**. [S.l.: s.n.], 1996. v. 96, p. 148–156.

GAIOSO, R. et al. Performance evaluation of single vs. batch of queries on gpus. **Concurrency and Computation: Practice and Experience**, Wiley Online Library, p. e5474, 2019.

GAMA, J. **Knowledge Discovery from Data Streams**. 1st. ed. [S.l.]: Chapman & Hall/CRC, 2010. ISBN 1439826110.

_____. A survey on learning from data streams: current and future trends. **Progress in Artificial Intelligence**, Springer, v. 1, n. 1, p. 45–55, 2012.

GAMA, J.; GABER, M. M. **Learning from Data Streams: Processing Techniques in Sensor Networks**. 1st. ed. [S.l.]: Springer, 2007. ISBN 978-3-540-73679-0.

GAMA, J.; RODRIGUES, P. P. An overview on mining data streams. In: _____. **Foundations of Computational, IntelligenceVolume 6: Data Mining**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. p. 29–45. ISBN 978-3-642-01091-0. Disponível em: <https://doi.org/10.1007/978-3-642-01091-0_2>.

GARCíA-MARTíN, E.; BIFET, A.; LAVESSON, N. Energy modeling of hoeffding tree ensembles. In: **Intelligent Data Analysis**. [S.l.]: IOS Press, 2021. v. 25, n. 1, p. 81–104.

GHOTING, A. et al. Nimble: A toolkit for the implementation of parallel data mining and machine learning algorithms on mapreduce. In: **Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining**. New York, NY, USA: Association for Computing Machinery, 2011. (KDD '11), p. 334–342. ISBN 9781450308137. Disponível em: <https://doi.org/10.1145/2020408.2020464>.

GOMES, H. M. et al. A survey on ensemble learning for data stream classification. **ACM Computing Surveys (CSUR)**, ACM New York, NY, USA, v. 50, n. 2, p. 1–36, 2017.

_____. Adaptive random forests for evolving data stream classification. **Machine Learning**, v. 106, n. 9, p. 1469–1495, 2017. ISSN 1573-0565. Disponível em: <https://doi.org/10.1007/s10994-017-5642-8>.

Gomes, H. M. et al. Feature scoring using tree-based ensembles for evolving data streams. In: **2019 IEEE International Conference on Big Data (Big Data)**. [S.l.: s.n.], 2019. p. 761–769. ISSN null.

GOMES, H. M.; READ, J.; BIFET, A. Streaming random patches for evolving data stream classification. In: **2019 IEEE International Conference on Data Mining (ICDM)**. [S.l.: s.n.], 2019. p. 240–249.

GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. **Deep Learning**. [S.l.]: MIT Press, 2016. <http://www.deeplearningbook.org>.

GUEGAN, L.; ORGERIE, A.-C. Estimating the end-to-end energy consumption of low-bandwidth iot applications for wifi devices. In: **CloudCom 2019 - 11th IEEE International Conference on Cloud Computing Technology and Science**. Sydney, Australia: IEEE, 2019. Disponível em: <https://hal.archives-ouvertes.fr/hal-02352637>.

HAJEWSKI, J.; OLIVEIRA, S. Distributed smsvm ensemble learning. In: ONETO, L. et al. (Ed.). **Recent Advances in Big Data and Deep Learning**. Cham: Springer International Publishing, 2020. p. 7–16. ISBN 978-3-030-16841-4.

HE, B. et al. Comet: batched stream processing for data intensive distributed computing. In: **Proceedings of the 1st ACM symposium on Cloud computing**. [S.l.: s.n.], 2010. p. 63–74.

HENNESSY, J. L.; PATTERSON, D. A. **Computer Architecture: A Quantitative Approach**. 5. ed. Amsterdam: Morgan Kaufmann, 2012. ISBN 978-0-12-383872-8.

HORAK, V.; BERKA, T.; VAJTERSIC, M. Parallel classification with two-stage bagging classifiers. **Computing and Informatics**, v. 32, p. 661–677, 01 2013.

HOYOS-IDROBO, A. et al. Frem – scalable and stable decoding with fast regularized ensemble of models. **NeuroImage**, v. 180, p. 160 – 172, 2018. ISSN 1053-8119. New advances in encoding and decoding of brain signals. Disponível em: <http://www.sciencedirect.com/science/article/pii/S1053811917308182>.

HUSSAIN, H. et al. An adaptive fpga implementation of multi-core k-nearest neighbour ensemble classifier using dynamic partial reconfiguration. In: **22nd International Conference on Field Programmable Logic and Applications (FPL)**. [S.l.: s.n.], 2012. p. 627–630.

IBRAHIM, K. Z.; STROHMAIER, E. Characterizing the relation between apex-map synthetic probes and reuse distance distributions. In: **2010 39th International Conference on Parallel Processing**. [S.l.: s.n.], 2010. p. 353–362.

ISLAM, R. et al. Spam filtering for network traffic security on a multi-core environment. **Concurr. Comput.: Pract. Exper.**, John Wiley and Sons Ltd., GBR, v. 21, n. 10, p. 1307 – 1320, jul. 2009. ISSN 1532-0626.

JACOB, B.; WANG, D.; NG, S. **Memory systems: cache, DRAM, disk**. [S.l.]: Morgan Kaufmann, 2010.

JAHNKE, G. **MRCRAIG: MapReduce and Ensemble Classifiers for Parallelizing Data Classification Problems**. Dissertação (Mestrado) — San Jose State University, 2009.

JAYASUNDARA, C.; GOPALAKRISHNAN, V. Facilitating multicast in vod systems by content pre-placement and multistage batching. In: IEEE. **2013 Fifth International Conference on Communication Systems and Networks (COMSNETS)**. [S.l.], 2013. p. 1–10.

JIN, R.; AGRAWAL, G. Communication and memory efficient parallel decision tree construction. In: **Proceedings of the 2003 SIAM International Conference on Data Mining**. [S.l.]: SIAM, 2003. (AAAIWS'93).

______. Efficient decision tree construction on streaming data. In: **Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining**. New York, NY, USA: Association for Computing Machinery, 2003. (KDD '03), p. 571–576. ISBN 1581137370. Disponível em: <https://doi.org/10.1145/956750.956821>.

JOSHI, M. V.; KARYPIS, G.; KUMAR, V. Scalparc: a new scalable and efficient parallel classification algorithm for mining large datasets. In: **Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing**. [S.l.: s.n.], 1998. p. 573–579.

KARP, R. M.; RAMACHANDRAN, V. A survey of parallel algorithms for shared-memory machines. 1 1989. Disponível em: <https://www.osti.gov/biblio/5805553>.

KENNEDY, K.; KOELBEL, C.; ZIMA, H. The rise and fall of high performance fortran: An historical object lesson. In: . New York, NY, USA: Association for Computing Machinery, 2007. (HOPL III), p. 7–1–7–22. ISBN 9781595937667. Disponível em: <https://doi.org/10.1145/1238844.1238851>.

KRAWCZYK, B. et al. Ensemble learning for data stream analysis: A survey. **Information Fusion**, v. 37, p. 132–156, 2017. ISSN 1566-2535. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1566253516302329>.

______. Ensemble learning for data stream analysis: A survey. **Information Fusion**, v. 37, p. 132–156, 2017. ISSN 1566-2535. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1566253516302329>.

KUKREJA, N. et al. Lossy checkpoint compression in full waveform inversion. **Geoscientific Model Development Discussions**, v. 2020, p. 1–26, 2020. Disponível em: <https://gmd.copernicus.org/preprints/gmd-2020-325/>.

LIAO, Y. et al. Learning random forests on the gpu. In: . [S.l.: s.n.], 2013.

LIU, X. **An Ensemble Method for Large Scale Machine Learning with Hadoop MapReduce**. Dissertação (Mestrado) — Electrical Engineering and Computer Science, University of Ottawa, 2014.

LOPES, J. F. et al. Evaluating the four-way performance trade-off for data stream classification in edge computing. **IEEE Transactions on Network and Service Management**, v. 17, n. 2, p. 1013–1025, 2020.

MAEDA, R. K. V. et al. Fast and accurate exploration of multi-level caches using hierarchical reuse distance. In: **2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)**. [S.l.: s.n.], 2017. p. 145–156.

MANAPRAGADA, C.; WEBB, G. I.; SALEHI, M. Extremely fast decision tree. In: **Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining**. [S.l.: s.n.], 2018. p. 1953–1962.

MARRóN, D. et al. Low-latency multi-threaded ensemble learning for dynamic big data streams. In: **IEEE International Conference on Big Data (BIGDATA)**. [S.l.: s.n.], 2017. p. 223–232.

MARTIN, E. G.; LAVESSON, N.; GRAHN, H. Energy efficiency in data stream mining. In: **2015 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)**. [S.l.: s.n.], 2015. p. 1125–1132.

MARTINOVIC, T. et al. On-line application autotuning exploiting ensemble models. In: . [S.l.: s.n.], 2019.

MASUD, M. et al. Classification and novel class detection in concept-drifting data streams under time constraints. **IEEE Transactions on Knowledge and Data Engineering**, v. 23, n. 6, p. 859–874, 2011.

MASUD, M. M. et al. Addressing concept-evolution in concept-drifting data streams. In: **2010 IEEE International Conference on Data Mining**. [S.l.: s.n.], 2010. p. 929–934.

MOORE, G. E. Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff. **IEEE Solid-State Circuits Society Newsletter**, v. 11, n. 3, p. 33–35, 2006.

NOJIMA, Y.; MIHARA, S.; ISHIBUCHI, H. Ensemble classifier design by parallel distributed implementation of genetic fuzzy rule selection for large data sets. In: **IEEE Congress on Evolutionary Computation**. [S.l.: s.n.], 2010. p. 1–8.

ORACLE. **Java Garbage Collection Basics**. 2021. Online. Acessed on: June 20th, 2021.

OZA, N. C.; RUSSELL, S. Online bagging and boosting. In: JAAKKOLA, T.; RICHARDSON, T. (Ed.). **Eighth International Workshop on Artificial Intelligence and Statistics**. Key West, Florida. USA: Morgan Kaufmann, 2001. p. 105–112.

PANDA, B. et al. Planet: Massively parallel learning of tree ensembles with mapreduce. **Proc. VLDB Endow.**, VLDB Endowment, v. 2, n. 2, p. 1426–1437, ago. 2009. ISSN 2150-8097. Disponível em: <https://doi.org/10.14778/1687553.1687569>.

PEREIRA, R. et al. Energy efficiency across programming languages: How do energy, time, and memory relate? In: **Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering**. New York, NY, USA: Association for Computing Machinery, 2017. (SLE 2017), p. 256–267. ISBN 9781450355254. Disponível em: <https://doi.org/10.1145/3136014.3136031>.

POLIKAR, R. Ensemble based systems in decision making. **IEEE Circuits and Systems Magazine**, v. 6, n. 3, p. 21–45, 2006.

PUHL, L. et al. Distributed novelty detection at the edge for iot network security. in press.

QIAN, Q. et al. Grid-based high performance ensemble classification for evolving data stream: Grid-based ensemble classification for data stream. **Concurrency and Computation: Practice and Experience**, v. 28, 08 2016.

SAFFARI, A. et al. On-line random forests. In: **2009 IEEE 12th International Conference on Computer Vision Workshops, ICCV Workshops**. [S.l.: s.n.], 2009. p. 1393 – 1400.

SCHöLKOPF, B.; PLATT, J.; HOFMANN, T. Map-reduce for machine learning on multicore. In: _____. **Advances in Neural Information Processing Systems 19: Proceedings of the 2006 Conference**. [S.l.: s.n.], 2007. p. 281–288.

SENGER, H. et al. Bsp cost and scalability analysis for mapreduce operations. **Concurrency and Computation: Practice and Experience**, Wiley Online Library, v. 28, n. 8, p. 2503–2527, 2016.

SHEN, X.; SHAW, J. Scalable implementation of efficient locality approximation. In: SPRINGER. **International Workshop on Languages and Compilers for Parallel Computing**. [S.l.], 2008. p. 202–216.

SILVA, J. A. et al. Data stream clustering: A survey. **ACM Computing Surveys (CSUR)**, ACM New York, NY, USA, v. 46, n. 1, p. 1–31, 2013.

SLUTZ, D. R.; TRAIGER, I. L. A note on the calculation of average working set size. **Commun. ACM**, Association for Computing Machinery, New York, NY, USA, v. 17, n. 10, p. 563–565, out. 1974. ISSN 0001-0782. Disponível em: <https://doi.org/10.1145/355620.361167>.

TAN, S. C.; TING, K. M.; LIU, T. F. Fast anomaly detection for streaming data. In: **Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume Two**. [S.l.]: AAAI Press, 2011. (IJCAI'11), p. 1511–1516. ISBN 9781577355144.

TSYMBAL, A. The problem of concept drift: Definitions and related work. 05 2004.

VALLE, C. et al. Parallel approach for ensemble learning with locally coupled neural networks. **Neural Processing Letters**, v. 32, p. 277–291, 12 2010.

VIEIRA, J.; DUARTE, R. P.; NETO, H. C. knn-stuff: knn streaming unit for fpgas. **IEEE Access**, v. 7, p. 170864–170877, 2019.

WANG, D. et al. Energy-optimal batching periods for asynchronous multistage data processing on sensor nodes: foundations and an mplatform case study. **Real-Time Systems**, Springer, v. 48, n. 2, p. 135–165, 2012.

WANG, J. Performance analysis of decision tree learning algorithms on multicore cpus. In: . [S.l.: s.n.], 2016.

WEILL, C. et al. Adanet: A scalable and flexible framework for automatically learning ensembles. In: **arXiv e-prints**. [S.l.: s.n.], 2019. p. arXiv:1905.00080.

WEN, Y.; TRAN, D.; BA, J. Batchensemble: an alternative approach to efficient ensemble and lifelong learning. In: **International Conference on Learning Representations**. [s.n.], 2020. Disponível em: <https://openreview.net/forum?id=Sklf1yrYDr>.

WITTEN, I. H.; FRANK, E. Data mining: practical machine learning tools and techniques with java implementations. **Acm Sigmod Record**, ACM New York, NY, USA, v. 31, n. 1, p. 76–77, 2002.

XAVIER, L.; THIRUNAVUKARASU, R. A distributed tree-based ensemble learning approach for efficient structure prediction of protein. **International Journal of Intelligent Engineering and Systems**, v. 10, p. 226–234, 03 2017.

YAN, R. et al. Large-scale multimedia semantic concept modeling using robust subspace bagging and mapreduce. In: **Proceedings of the First ACM Workshop on Large-Scale Multimedia Retrieval and Mining**. New York, NY, USA: Association for Computing Machinery, 2009. (LS-MMRM '09), p. 35–42. ISBN 9781605587561. Disponível em: <https://doi.org/10.1145/1631058.1631067>.

YUAN, L. et al. A relational theory of locality. **ACM Transactions on Architecture and Code Optimization (TACO)**, ACM New York, NY, USA, v. 16, n. 3, p. 1–26, 2019.

ZAHARIA, M. et al. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In: USENIX ASSOCIATION. **Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing**. [S.l.], 2012. p. 10–10.

ZHANG, D. et al. Edgebatch: Towards ai-empowered optimal task batching in intelligent edge systems. In: **2019 IEEE Real-Time Systems Symposium (RTSS)**. [S.l.: s.n.], 2019. p. 366–379.

ŽLIOBAITė, I. et al. Evaluation methods and decision theory for classification of streaming data with temporal dependence. **Machine Learning**, v. 98, 03 2014.