Balancing Performance and Energy Consumption of Bagging Ensembles for the Classification of Data Streams in Edge Computing

Guilherme Cassales, Heitor Murilo Gomes, Albert Bifet, *Member, IEEE*, Bernhard Pfahringer, and Hermes Senger

***Abstract*—In recent years, the Edge Computing (EC) paradigm has emerged as an enabling factor for developing technologies like the Internet of Things (IoT) and 5G networks, bridging the gap between Cloud Computing services and end-users, supporting low latency, mobility, and location awareness to delay-sensitive applications. An increasing number of solutions in EC have employed machine learning (ML) methods to perform data classification and other information processing tasks on continuous and evolving data streams. Usually, such solutions have to cope with vast amounts of data that come as data streams while balancing energy consumption, latency, and the predictive performance of the algorithms. Ensemble methods achieve remarkable predictive performance when applied to evolving data streams due to several models and the possibility of selective resets. This work investigates a strategy that introduces short intervals to defer the processing of mini-batches. Well balanced, our strategy can improve the performance (i.e., delay, throughput) and reduce the energy consumption of bagging ensembles to classify data streams. The experimental evaluation involved six state-of-art ensemble algorithms (OzaBag, OzaBag Adaptive Size Hoeffding Tree, Online Bagging ADWIN, Leveraging Bagging, Adaptive RandomForest, and Streaming Random Patches) applying five widely used machine learning benchmark datasets with varied characteristics on three computer platforms. As a result, our strategy can significantly reduce energy consumption in 96% of the experimental scenarios evaluated. Despite the trade-offs, it is possible to balance them to avoid significant loss in predictive performance.**

***Index Terms*—Edge computing, machine learning, data stream classification, ensembles, energy consumption.**

Manuscript received 17 December 2021; revised 18 May 2022; accepted 23 November 2022. Date of publication 5 December 2022; date of current version 9 October 2023. This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001, and Programa Institucional de Internacionalização – CAPES-PrInt UFSCar (Contract 88887.373234/2019-00). Authors also thank Stic AMSUD (project 20-STIC-09), and FAPESP (contract numbers 2019/26702-8, and 2021/00199-8) for their support. Partially supported by the TAIAO project CONT-64517-SSIFDS-UOW (Time-Evolving Data Science / Artificial Intelligence for Advanced Open Environmental Science) funded by the New Zealand Ministry of Business, Innovation, and Employment (MBIE). URL: https://taiao.ai/. The associate editor coordinating the review of this article and approving it for publication was J. Sa Silva. *(Corresponding author: Hermes Senger.)*

Guilherme Cassales, Albert Bifet, and Bernhard Pfahringer are with the Department of Computer Science, University of Waikato, Hamilton 3240, New Zealand.

Heitor Murilo Gomes is with the School of Engineering and Computer Science, Victoria University of Wellington, Wellington 6140, New Zealand.

Hermes Senger is with the Department of Computer Science, Federal University of São Carlos, São Carlos 13565-905, Brazil (e-mail: senger.hermes@gmail.com)

Digital Object Identifier 10.1109/TNSM.2022.3226505

I. Introduction

EDGE Computing (EC) is an enabling paradigm for developing technologies like the Internet of Things (IoT), 5G, online gaming, augmented reality (AR), vehicle-to-vehicle communications, smart grids, and real-time video analytics. In essence, EC brings the services and utilities of cloud computing closer to the end-user, providing low latency, location awareness, and better efficiency for delay-sensitive mobile applications [1]. Computational resources are placed close location to the mobile devices, with moderate capability servers placed at the edge of the network to achieve necessary user-centric requirements [2].

Many applications like smart grids with a large number of sensors continuously collect data from the surrounding environment across medium to large geographical areas [3]. Because most sensors are resource-constrained, they cannot run expensive or data-intensive tasks like machine learning (ML) algorithms. In this case, it is more efficient to send these data to more powerful resources nearby (placed on the edge of the network) for faster processing (i.e., with low latency) [4], [5]. Another example is wearable technologies, which can depend on energy- and resource-constrained mobile devices or low-end servers placed nearby to process their data streams [6].

In these scenarios, a remarkable trend is the increasing adoption of ML techniques to execute tasks like data classification, spam filtering, anomaly detection in network traffic for cybersecurity, real-time image classification and segmentation, driving support applications, autonomous driving vehicles, and many others. However, the need to cope with these dynamic environments with potentially infinite data streams with non-stationary behavior and run ML algorithms on mobile devices or mid-end servers on edge creates additional challenges. Adapting the optimization techniques for different types of hardware may be cumbersome and time-consuming.

Historically, ML research focused on improving predictive performance without computational resources and energy consumption constraints. However, on the algorithmic side, the ML field is shifting towards data stream learning to face the challenges above, where requirements like single pass, response time, and constant memory usage are imposed [7]. One example of this phenomenon among ML techniques is the ensemble technique. Ensembles have demonstrated remarkable predictive performance for the classification of data streams [8]

at the expense of higher resource consumption and increased latency.

Nevertheless, little effort has been made to reduce the energy consumption of data stream algorithms [9]. To address this gap, we have extended previous research [10], [11] to investigate how to optimize the classification of data streams with bagging ensembles concerning energy efficiency, time performance (i.e., delay and throughput), and predictive performance. In [10], [11], we proposed the use of mini-batching as a technique for improving the efficiency of resource utilization of ensembles for processing data streams. In [11], we proved that mini-batching provides optimal memory locality for data stream ensembles and studies the trade-off between time and predictive performance. However, we neither addressed the use of the mini-batching to reduce energy consumption nor the balance between energy consumption, predictive performance, and time cost.

The main contributions of this paper can be summarized as follows:

1) Present the premier study that applies a strategy of deferred mini-batching (which extends the *mini-batching* proposed in [10], [11]) for improving the performance and energy efficiency of bagging ensembles. Our results present significant gains in 96% of the evaluated cases.
2) We extend the literature's energy model for bagging ensembles to contemplate mini-batching. In addition, we identify the trade-offs between time performance, energy efficiency, and predictive performance of such algorithms in online data streams;
3) We demonstrate how to achieve a good balance of these trade-offs to obtain significant gains in time performance and energy efficiency at the cost of negligible loss in predictive performance on a realistic stream processing testbed using five widely popular datasets and different levels of workload intensity;
4) We also advance the literature on evaluating the energy consumption for data stream classification by testing six state-of-art bagging ensemble algorithms and using three computer architectures ranging from micro- to middle-end servers.

The remainder of this article is organized as follows, Section II discusses the related works. Six state-of-art bagging ensemble algorithms are described in Section III. We present our proposal for applying mini-batching for parallel implementations of bagging ensemble algorithms in Section IV, followed by the experimental evaluation in Section V. Finally, our conclusion is presented in Section VII.

II. Related Work

Computing nodes, from smallest devices to high-end servers, comprise several components, most of which can be optimized to save energy [12]. From atomic block components like a simple, functional unit within a chip to CPU cores, disks, network interface cards (NICs), and entire boards can be put in sleep or idle states to save power when not delivering services to achieve proportional computing [13]. Dynamic power management (DPM) encompasses a set of techniques that achieve energy-efficient by selectively turning off (or reducing the performance of) system components when they are idle (or partially unexploited) [14]. Dynamic Voltage and Frequency Scaling (DVFS) defines several levels of frequency at which a processor can operate, where lower frequencies become slower to save power [12], [15]. Sensors and wearable devices that use small-sized batteries can also be optimized to save power while delivering functionalities including sensing, storage, and computation [6]. The work in [5] combines the use of DVFS and DPM for optimizing performance and energy savings of single-board computers using Pareto frontier for multi-criteria optimization of several computing-intensive kernels. Although DVFS can reduce energy consumption, scaling the CPU clock frequency also affects the performance of other applications running in the same node.

A modular, scalable, and efficient FPGA-based implementation of kNN for System on Chip devices is presented in [16]. The solution shows improvements of 60X in execution time and 50X in energy efficiency due to the low power consumption of FPGAs. Despite its outstanding performance, the contribution is both algorithm and hardware-specific.

The work in [17] emphasizes energy consumption and energy efficiency as essential 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.

In [18], the authors analyze power consumption for both batch and online data stream learning. They 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.

Aiming to reduce the memory cost, the work in [19] proposed the Strict VFDT (SVFDT), an extension of the VFDT. Designed for memory-constrained devices, SVFDT reduces memory usage and execution time by minimizing tree growth while maintaining competitive predictive performance. Further, the work assessed energy cost by tuning the hyper-parameters of VFDT and SVFDT in [20], and for them and SVFDT with Online Local Boosting (OLBoost) in [4]. The works in [20] and [4] discuss the performance trade-offs involving time cost, memory, energy consumption, and predictive performance for the execution of ML algorithms in EC. They also demonstrate that the most complex method delivers the best predictive performance at the expense of worse memory and energy performance.

In [9], the authors presented an energy-efficient approach to real-time prediction with high levels of accuracy called *nmin adaptation*, which reduces the energy consumption of Hoeffding Trees ensembles by adapting the number of instances required for a node 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. Although these works propose energy-efficient algorithms, the benefit achieved in these works focuses on specific algorithms. In contrast, our work focus on optimizations that can benefit several bagging ensembles.

In summary, the studies on energy consumption are still scarce and mainly focus on designing power-efficient algorithms and data structures or optimizing existing ones. This paper proposes a strategy that employs deferred mini-batching to reduce the power consumption of ensembles of classifiers to process data streams. The rationale behind this strategy is to interleave the processing of mini-batches with short periods of CPU idleness (of milliseconds), which can trigger DPM techniques[1] deployed in modern CPUs in order to save energy. The strategy consists of controlling the mini-batch size to increase the time interval between the processing of mini-batches, thus saving more energy. On the other hand, larger batches will increase the latency for processing data streams and decrease the predictive performance of ensembles of classifiers. In the present article we present a study on the proper sizing of mini-batches and duration of idle periods in between can thus balance energy consumption, latency, and predictive performance of such algorithms. However, the implementation of a control loop to automatically set the mini-batch size is out of the scope of this work.

Finally, in essence, mini-batching consists of applying a loop-exchange optimization (by iterating over the ensembles in the outermost loop and then over the data instances in the innermost loop). As mini-batching does not change the implementation of the individual learners, it can wrap around other individual learners, along with their algorithm-specific optimizations (e.g., in [9], [16], [17], [19]). This, however, is out of this work's scope.

III. Bagging Ensembles for Stream Processing

Learning algorithms have to cope with dynamic environments that collect potentially unlimited data streams in many applications. 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 (n$\longrightarrow \infty$) [21]. As mentioned before, stream processing algorithms have additional requirements, which may be related to memory, response time, or a transient behavior presented by the data stream. In this context, one of the most widely used algorithms is the Hoeffding Tree [22].

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 observed variable is at least within $\pm\epsilon$ of the average.

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

[1]Dynamic power management strategies.

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 is that a single tree is considered a 'weak' model because a single tree cannot accurately model complex learning problems. One approach to circumvent such 'weakness' is to ensemble several models. A popular strategy to create an ensemble of learners is Bagging [23]. Although Breiman proposed Bagging and its variants (e.g., Random Forest) more than 20 years ago [23], they are still used to this day as an effective method to reduce error without resorting to intricate models, such as deep neural networks, that are not trivial to train and fine-tune. In contrast to Boosting [24], Bagging does not create dependency among the base models, facilitating the parallelization of the method in an online fashion. Fig. 1 depicts the streaming adaptation of Bagging that calculates different weights using a Poisson distribution to simulate the batch bootstrapping process and preserves independence among the learners. The Poisson distribution effectively creates several different subsets of the data that allow repeatedly training with some instances while ignoring others in the training phase.

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 [25], [26], [27], [28]. This behavior can be attributed to the difficulty of effectively modelling the dependencies in a streaming scenario, as noted in [8].

Next, we present a summary description of six ensemble algorithms that evolved from the original Bagging to an online (streaming) setting by Oza and Russell [25]. Despite other decision tree algorithms [29], the HT algorithm is often chosen as the base model for the online bagging algorithms. The HT is often a common denominator. It may not be the most accurate individually, but it does yield good predictive performance without requiring too many computational resources. Our experiments employ the HT as the base model for all six algorithms used to evaluate our mini-batching strategy.

Online Bagging (OzaBag - OB) [25] is an incremental adaptation of the original Bagging algorithm. The authors demonstrate how the process of bootstrapping can be adapted to an online setting using a Poisson ($\lambda = 1$) distribution. In essence, instead of sampling with replacement from the original training set, in Online Bagging, the Poisson ($\lambda = 1$) is used to assign weights to each incoming instance. These weights represent the number of times an instance will be 'repeated' to simulate bootstrapping. One concern with using $\lambda = 1$ is that about 37% of the instances will receive weight 0, thus not being used to train, which is desired to approximate it to the offline version of Bagging but may be detrimental to an online learning setting [8]. Therefore, other works [27], [28] increase the number of times an instance is used for training by increasing the λ parameter. We chose this algorithm because it can be considered a baseline for bagging ensembles. Furthermore, it applies bagging without any additional measures to improve the result. Thus, it is the fastest algorithm,

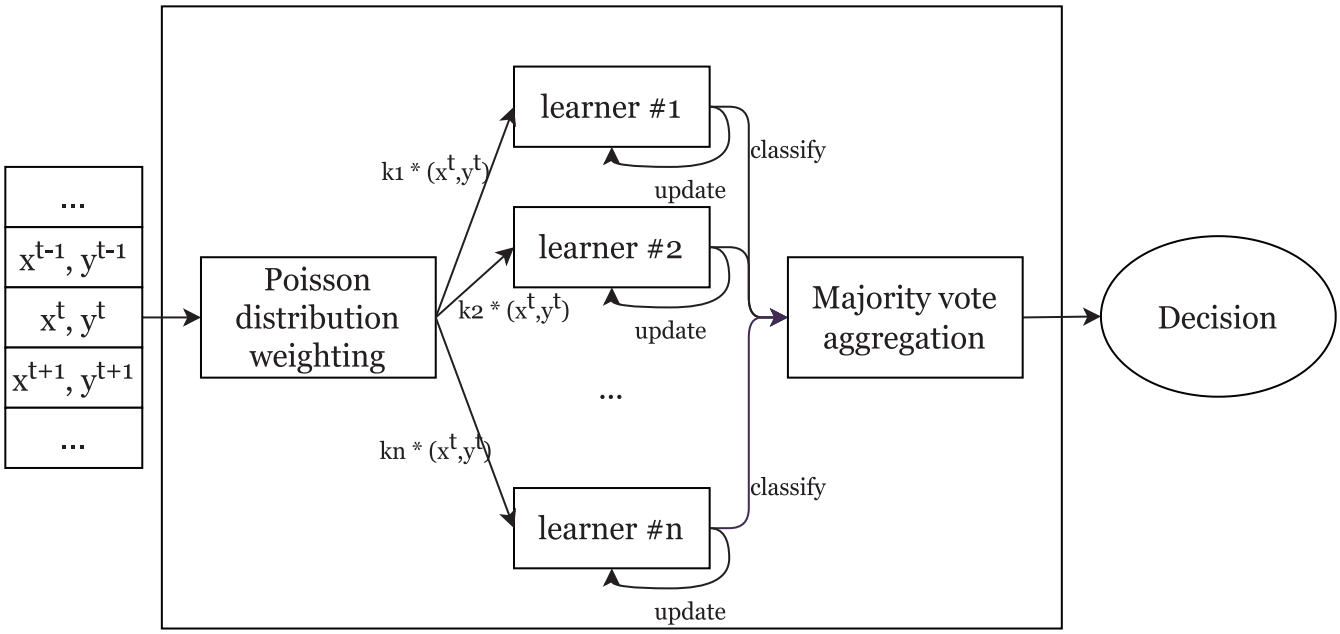


Fig. 1. Example of a Bagging Ensemble organization.

with the least computational cost and the worst predictive performance.

OzaBag Adaptive Size Hoeffding Tree (OBagASHT) [26] combines the OzaBag with Adaptive-Size Hoeffding Trees (ASHT). The new trees have a maximum number of split nodes and some policies to prevent the tree from growing bigger than this parameter (i.e., deleting some nodes). This algorithm's objective was to improve predictive performance by enforcing the creation of different trees. Effectively, diversity is created by having different reset-speed trees in the ensemble, according to the maximum size. The intuition is that smaller trees can adapt more quickly to changes, and larger trees can provide better 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 [8]. This algorithm creates tasks with heterogeneous workloads. For example, when scheduling 100 parallel tasks on a smaller number of cores, the difference in tree size and, consequently, training time per instance between the tasks make it difficult to synchronize efficiently. This challenge turns this algorithm into an interesting use case to evaluate mini-batching.

Online Bagging ADWIN (OBADWIN) [26] combines OzaBag with the ADAptive WINdow (ADWIN) [30] change detection algorithm. When a change is detected, the classifier with the lowest predictive performance is replaced by a new classifier. 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. This implies that the average over the existing window can be reliably taken as an estimation of the current average in the stream, except for a very small or very recent change that is still not statistically visible. This algorithm represents a trade-off between the ever-increasing computational cost of the trees that keep growing and the addition of a change detector that resets trees according to the stream behavior. This trade-off implies that OBADWIN will initially have a higher computational cost but eventually benefit from the change detector as OzaBag's trees grow.

Leveraging Bagging (LBag) [27] extends OBADWIN by increasing the λ parameter of the Poisson distribution to 6, effectively causing each instance to have a higher weight and be more used for training more often. In contrast to OBADWIN, LBag maintains one ADWIN detector per model in the ensemble and independently resets the models. This approach leverages the predictive performance of OBADWIN by merely training each model more often (higher weight) and resetting them individually. One drawback of LBag compared with OB and OBADWIN is that it requires more memory and processing time since the base models are trained more often, and there are more instances of ADWIN. In [27], the authors also attempted to further increase the diversity of LBag by randomizing the output of the ensemble via random output codes. However, this approach was not very successful compared to maintaining a deterministic combination of the models' outputs. LBag constitutes another use case where the tasks can become very heterogeneous. For example, making trees grow faster and adding a change detector makes it possible to have tasks with large trees that take much time to train and smaller trees with fast training time. Although LBag is similar to use cases OBagASHT and OBADWIN, it is much more volatile and dynamic.

Adaptive Random Forest (ARF) is an adaptation of the original Random Forest algorithm [31] to the streaming setting. Random Forest can be seen as an extension of Bagging, where further diversity among the base models (decision trees)

is obtained by randomly choosing a subset of features to be used for further splitting leaf nodes. ARF uses the incremental decision tree algorithm Hoeffding tree [22] and simulates resampling 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 a warning is signalled, another model is created (a 'background tree') and trained without affecting the ensemble predictions. If the warning escalates to a drift signal, the associated tree is replaced by 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 [28], the co-existence of a tree and its background tree is often short-lived. ARF increases the computation of each tree and eliminates the cases where recently reset trees will be much faster than the rest of the ensemble. The downside is an overall higher cost of maintaining background trees. This use case represents when the tasks are more homogeneous and costly.

Streaming Random Patches (SRP) [32] is an ensemble method specially adapted to 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, i.e., SRP uses global randomization while ARF uses local randomization. Even though, in [32] all the experiments focused on Hoeffding trees and showed that SRP could produce deeper trees, which may lead to increased diversity in the ensemble. SRP takes the computational cost a step further compared to LBag and ARF. Creating deeper trees and using global randomization can increase the training time. Therefore, creating the heaviest workload across all algorithms.

IV. Mini-Batching as a Strategy for Improving Time and Energy Performance of Ensembles

Although all the learners who compose an ensemble may be homogeneous in type, each has its own (and different) model. For instance, all learners implement a Hoeffding Tree that grows in a different shape and can change over time. One advantage of such methods is that task parallelism can naturally be applied as the underlying classifiers in bagging ensembles execute independently from each other and without communication.

Algorithm 1 depicts a task-parallel-based implementation. This version improves the performance of the current parallel implementation of the ARF algorithm [28], in the latest version in MOA [33], by reusing the data structures and avoiding the costs of allocating new ones for every instance to be processed.

In lines 2-3, we start a thread pool and create one Trainer (runnable) for each ensemble classifier. For each arriving data instance (lines 4-17), if the program's elapsed time has not surpassed the set timeout, we obtain the votes from all the classifiers (line 5). Then, we compute the *Poisson* weights and update the data structures for training in lines 6-9. The prediction phase has a low computational cost because the algorithm uses Hoeffding trees [22], and thus, we classify instances sequentially. On the other hand, the training phase is more expensive. It involves updating many statistics on each tree's nodes, calculating new splits, and detecting data distribution changes (for three methods). As the training phase dominates the computational cost, parallelism is implemented (in lines 10-13) by simultaneously training many classifiers. Lines 14-16 represent the global change detector, present only on OBAdwin and OBagASHT, where we replace the ensemble's worst classifier with a brand new one. In the case of the algorithms with one change detector per weak classifier, this line would be internal to each classifier. Therefore, when a change is detected, it resets the classifier itself (as shown in Algorithm 3). Finally, lines 17-19 represent the timeout condition, where the ensemble will run for a limited period and then finish processing.

Algorithm 1 Task-Based Parallel Implementation

```
1:  Input: an ensemble E, num_threads, a data stream S
2:  P ← Create_service_thread_pool(num_threads)
3:  T ← 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 ← poisson(λ)
8:          T_i.update(I, k)
9:      end for
10:     for all trainers T do in parallel
11:         W_inst ← I * k
12:         Train_on_instance(W_inst)
13:     end for
14:     if change detected then
15:         reset_worst_weak_classifier
16:     end if
17:     if ElapsedTime > Timeout then
18:         break
19:     end if
20: end for
```

A. Deferred Mini-Batching

Although task parallelism looks straightforward for implementing ensembles, poor memory usage can severely hinder their performance. For instance, high-frequency access to data structures larger than cache memories can raise performance bottlenecks. Also, algorithms that continuously perform memory allocation/release operations to discard old models and create new ones during the learning/training process may pressure the garbage collection. To mitigate such problems, in a previous work we introduced mini-batching [11], an optimization strategy which groups several data instances of a stream for processing. The algorithm assigns a task to each learner. The tasks' responsibility is to process training by iterating uninterruptedly through all instances of a mini-batch instead of processing a single instance at a time. When a task is invoked, its data structures are loaded into the upper levels of the memory hierarchy (upper-level caches). Once on the upper-level caches, the data structures can be quickly accessed to process the remaining

Algorithm 2 The Deferred Mini-Batching

```
1:  Input: an ensemble E, num_threads, a data stream S, mini-batch size L_mb
2:  P ← Create_service_thread_pool(num_threads)
3:  T ← Create_trainers_collection(E)
4:  for each arriving instance I in stream S do
5:      if ElapsedTime > Timeout then
6:          E.process_minibatch(B)
7:          break loop
8:      end if
9:      B.append(I)
10:     if B.size() == L_mb then
11:         E.process_minibatch(B)
12:     end if
13:     sleep();            ▷ Sleep until next instance arrives
14: end for
```

Algorithm 3 process_minibatch Routine

```
1:  Input: mini-batch B
2:  for each trainer T_i in trainers T do in parallel
3:      T_i.instances ← B
4:      votes_i ← T_i.classify(T_i.instances)
5:  end for
6:  E.compile(votes)
7:  for each trainer T_i in trainers T do in parallel
8:      for each instance I in T_i.instances do
9:          k ← poisson(λ)
10:         W_inst ← I * k
11:         T_i.train_on_instance(W_inst)
12:     end for
13:     if change detected then
14:         reset_classifier
15:     end if
16: end for
17: B.clear()
```

instances of the same mini-batch, reducing cache misses and improving performance.

The Algorithm 2 shows the mini-batching strategy. The first difference between the two algorithms appears in lines 5-10 of the Algorithm 2, where the ensemble will only accumulate the instances until the desired mini-batch size is met, the time limit is reached, or the stream ends. If the time limit is reached (line 5), the algorithm treats the current mini-batch as if it was complete, processes it, and breaks the execution loop. In normal cases, when the mini-batch size reaches the size set as a parameter (line 10), the algorithm processes the mini-batch. So far, the deferred mini-batching behaves similarly to the original mini-batching proposed in [10], [11]. However, the *sleep* operation in line 13 introduces a new feature which the releases the processor a new mini-batch is filled up with arriving instances. The strategy is grounded on the assumption that the ensemble is presented one instance of the data stream at a time. In real implementations, such a sleep operation is implicitly implemented by I/O subsystems (e.g., when the algorithm invokes a blocking read operation on a socket to wait for a new incoming data instance). The sleeping period was made explicit here because it is important twofold: (*i*) it releases CPU to other applications (running on the same edge nodes) while waiting for new arriving data instances; (*ii*) it creates opportunity for DPM strategies to turn off idle subsystems (e.g., CPU components) to save energy.

Algorithm 3 depicts the routine used to process the mini-batch. It performs the classification (lines 2-6) and training (lines 7-16). In line 3, we copy the whole mini-batch to each trainer. Line 4 uses the instances to compute votes for each trainer and stores them for later use. The votes are aggregated and compiled in line 6 to provide the predictions. The for in line 2 may be sequential or parallel according to the characteristics of the application (e.g., classifiers with small number of operations may disable the parallelism and run it sequentially). Then, each trainer will iterate (sequentially) through all mini-batch instances while calculating the weight, creating the weighted instance, and training the classifier with this instance (lines 7-12). ARF, SRP, and LBag, exclusively, will execute lines 13-15 as a local change detector for each classifier in the ensemble. In OBAdwin, lines 13-15 would be outside the parallel section, as the change detection is a global operation. Finally, in line 17, the mini-batch is emptied to begin accumulating again.

In essence, the grouping of instances and reordering of operations improve the algorithm's execution time and energy consumption thanks to a better access locality. Among the definitions of access locality provided by Yuan et al. [34], the definition of *reuse distance* (RD) can be used to demonstrate how the mini-batching approach can improve ensemble implementations' access locality. Reuse distance (RD) is defined as "the number of distinct data accessed since the last access to the same datum, including the reused datum" [34].

The RD is ∞ for its first access. The minimum RD is one since it includes the reused datum. The maximum RD, denoted by m, is the number of distinct data elements in the problem. In our case, m denotes the number of classifiers in the ensemble (ensemble size). For example, assuming a set of only three elements *a, b, c*, the RD sequence would be $\infty\infty\infty$ 333 333 for the access sequence *abc abc abc*. As another example, the RD sequence would be $\infty\infty\infty$ 135 135 for the access sequence *abc cba abc*. Therefore, disregarding the ∞ accesses, we can estimate the total RD for the sequential approach as follows:

$$RD_{sequential} = \sum_{1}^{n} \sum_{1}^{m} m \tag{2}$$

Using the same three-element set from the previous example, the access sequence when using mini-batching would change to *aaa bbb ccc*. In turn, the RD sequence would change to ∞ 11 ∞ 11 ∞ 11 for the first mini-batch and m 1 1 for all the subsequent mini-batches. Again, disregarding the ∞ accesses, we can estimate the total RD for the mini-batching strategy as follows:

$$RD_{mini-batching} = \sum_{1}^{n/b} \sum_{1}^{m} (m + b - 1) \tag{3}$$

As can be noted by comparing equations (2) and (3), the most significant contribution of the mini-batching strategy in reducing the RD occurs in the outer sum. By dividing the higher limit by the mini-batch size, a mini-batch size as small as ten can reduce the order of magnitude of this problem. For more information on this topic, refer to a previous work [11].

In summary, by reducing the RD, mini-batching reduces the number of CPU cycles (and ultimately the energy consumption) to process each data instance. However, notice that our strategy can improve energy efficiency in a second way. The sleep operation in line 4 of Algorithm 13 releases the CPU while waiting for the next data instance to arrive. Although this semantics can be implicit in many mini-batching implementations (e.g., our implementation blocks reading data instances from a socket), it was explicitly included in our algorithm description. These sleep periods put system components in idle state so letting DPM techniques implemented in modern processors to turn off components in order to save energy. Besides saving energy, sleep periods free the processor, which is useful to benefit other applications running on the same physical nodes in the edge.

V. Experimental Setup

This section describes the experimental evaluation of our proposal.

A. The Testbed

As EC implementations can encompass different hardware platforms ranging from small, low-end devices to mid-end computing servers, we assessed the optimizations proposed on three hardware architectures. The first hardware is a single board computer Raspberry Pi 3 Model B with a Broadcom BCM2837 processor of 4 cores Cortex-A53 64-bit SoC@1.2GHz and 1GB LPDDR2 SDRAM memory, which is frequently pointed as a representative platform for EC implementations. We also included two more powerful hardware platforms. A regular personal computer with Intel i5-2400 CPU@3.10GHz and 4GB memory. And a SYS-7049GP-TRT SuperServer Tower/4U Rackmountable with a dual-socket Intel Xeon Xeon CLX-SP 4208 8C/16T, 2200W Redundant Power Supplies, typically used in small enterprise or data center environments. The three hardware specs are described in Table I. We carried out the experiments on a testbed composed of four nodes (as depicted in Fig. 2 connected by a dedicated network.

TABLE I
Hardware Specifications

Machine type	mid-end server	personal computer	single-board computer
Processor	Intel Xeon 4208	Intel i5-2400	Broadcom BCM2835
Micro architecture	Cascade Lake	Sandy Bridge	Cortex-A53
Cores/socket	8	4	4
Threads/core	2	1	1
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	-
TDP	85 W	35 W	4 W

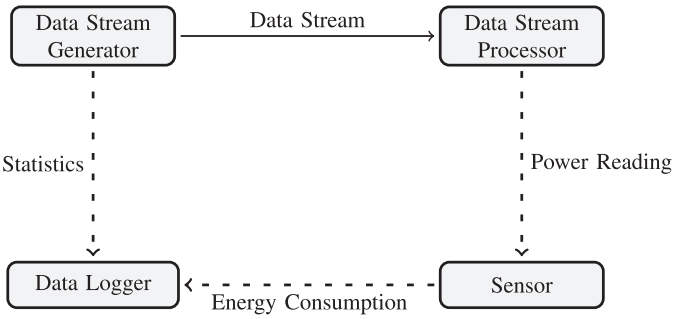


Fig. 2. The testbed is composed of four nodes (clockwise order): (a) a data stream (load) generator; (b) a data stream processor (whose architectures are described in Table I; (c) a high precision power meter (sensor); and (d) a data logger which registers all experimental data.

TABLE II
Summary of Dataset Statistics

Datasets	Airlines	GMSC	Electricity	Covertype	Kyoto
# Instances	540k	150k	45k	581k	725k
# Features	7	10	8	54	12
# Nominal feat	4	0	1	45	0
Normalized	No	No	Yes	Yes	Yes

B. Load Generation

The *data stream generator* reads the benchmark dataset and transmits the data samples over the network to the *data stream processor* node at controlled transmission rates (e.g., to generate low, moderate, and high workloads for each CPU type). We used five open access[2] datasets (whose characteristics are summarized in Table II) in the experiments:

- 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 U.S. 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.

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

- 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.

C. The Ensembles Used for Benchmarking

The *data stream processor* implements the optimizations as a wrapper for six ensemble algorithms described in section III. We implemented this module in the Massive Online Analysis (MOA) framework [33].[3] We adapted MOA to read from a socket instead of reading from a local ARFF file. We chose MOA because: (*i*) it provides correct and validated implementations of the six ensemble learners used in our experiments, (*ii*) MOA is straightforward to extend or modify, which allowed us to code mini-batching as a wrapper method for the uniform evaluation of the six ensembles with the optimizations; and (*iii*) there is a large body of research using MOA in the ML area [33], which lends high reproducibility to our study. The *data stream processor* is executed on each different hardware to evaluate its performance and power consumption.

Although Java does not focus on implementing either energy-efficient or high-performance applications, the work in [35] shows that Java is in the top 5 languages (out of 27 tested) that need less energy and time to execute the applications.

D. Performance and Power Consumption Measurements

The *data logger* (Fig. 2) collects all experimental data regarding the performance (e.g., throughput, processing delay) and the power consumed by the data stream processor for further analysis. The *sensor* is implemented by a high precision power meter (Yokogawa MW-100) which periodically collects information directly from the Power Distribution Unit (PDU) and sends to the data logger.

Our interest is to measure the amount of Energy (E) consumed to perform classification tasks. However, most electricity consumption monitors operate by collecting an instantaneous rate of Power (P) being supplied. Energy is the product of the average power and Time (t):

$$E = P \times t \tag{4}$$

Since power can vary in time, the total amount of energy consumed to perform a task is given by

$$E = \int_0^t P(t)\, dt$$

where t is the time to perform one task. In practice, we can obtain an approximation of E by taking several periodic measures:

$$E = \frac{1}{n} \sum_{i=1}^{n} P_i \times t, \tag{5}$$

where n is the number of samples taken by the monitor. Typically, *energy efficiency* is defined as the ratio of the energy spent and the amount of computing performed. As energy is expressed in Joules (J), and the work is expressed as the number of data instances processed (I), we can estimate energy efficiency (as Joules per Instance - JPI) by:

$$JPI = \frac{E}{I}. \tag{6}$$

In some experiments, we use performance metrics of *throughput*, given by the average number of data instances processed by second (IPS), and *delay*, which is the average time taken to perform the processing of data instances, including its the transmission over the network, the composition of mini-batches, and the time to process the whole mini-batch by the *data stream processor*.

The previous metrics are related to computational performance. A second dimension of performance we considered is the *predictive performance*, which can be hindered by the optimization techniques. *Accuracy* is a widely known measure and represents the percentage of correctly classified instances.

VI. Experimental Results and Analysis

As memory constraints of small devices can hinder the execution of large ensembles, our preliminary experiment aimed to find the influence of the ensemble size (i.e., the number of classifiers in the ensemble) on the accuracy, energy consumption, and throughput. For this experiment, we used the baseline version of the algorithms while measuring energy consumption. Results shown in Fig. 3 demonstrate that accuracy (in red) remains almost constant (less than 1% loss of accuracy), the throughput (in blue) decreases, and energy consumption (in green) increases as we increase the ensemble size (x-axis). Because accuracy loss is negligible for all datasets and all algorithms, used small ensembles in the Raspberry Pi experiments.

A second preliminary experiment aims at profiling the power consumption of each machine type as we increase the number of CPU cores running at full load. We used the `stress` application and thread pinning to fully load cores during 180 seconds. Results in Fig. 4 confirm our expectation for the RaspBerry Pi and Core i5-2400. Although the TDP for the Xeon 4208 is only 85 W,[4] the valued measured is higher because it refers to the whole machine (including disk, dual power supply, dual socket, 4 fans, etc.).

A. Energy Consumption

The next experiment evaluates the energy efficiency of algorithms running on the three machines under three workload intensities (i.e., at 10%, 50%, and 90% of its maximum throughput). To accomplish that, we used adapted MOA to read data instances from a socket instead of an ARFF file as described in Section V-C. Then, we tuned the load generator to deliver instances at rates equivalent to 10%, 50%, and 90% of the maximum capacity of the machine for each scenario. Experiments executed each scenario for at least 3 minutes in

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

[4] https://ark.intel.com/content/www/us/en/ark/products/193390/intel-xeon-silver-4208-processor-11m-cache-2-10-ghz.html

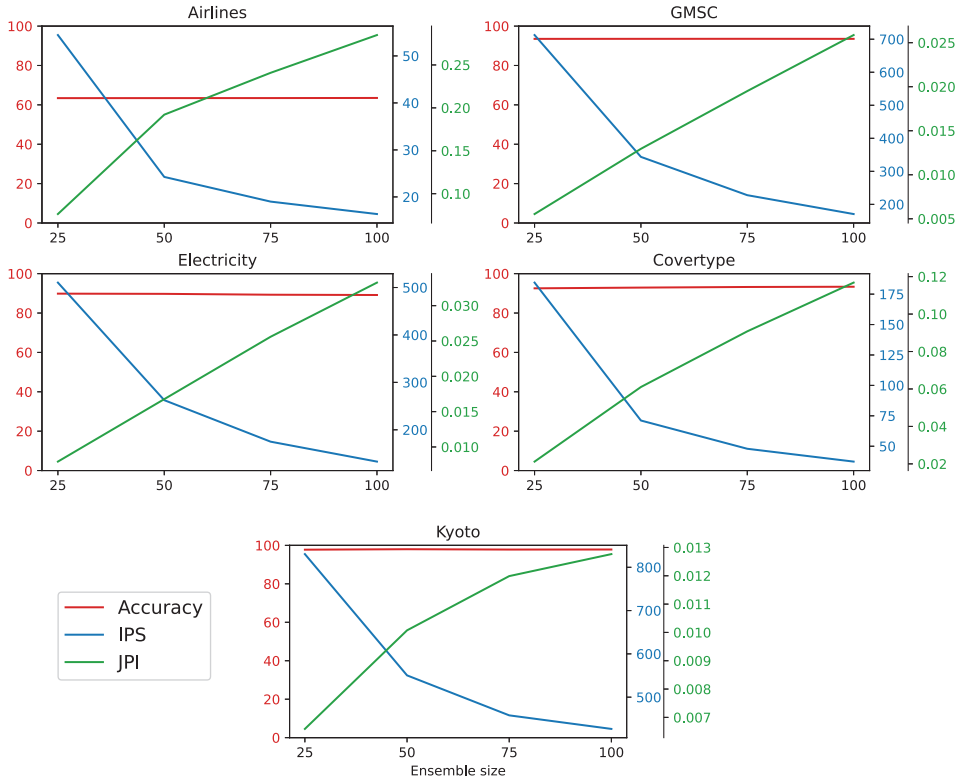
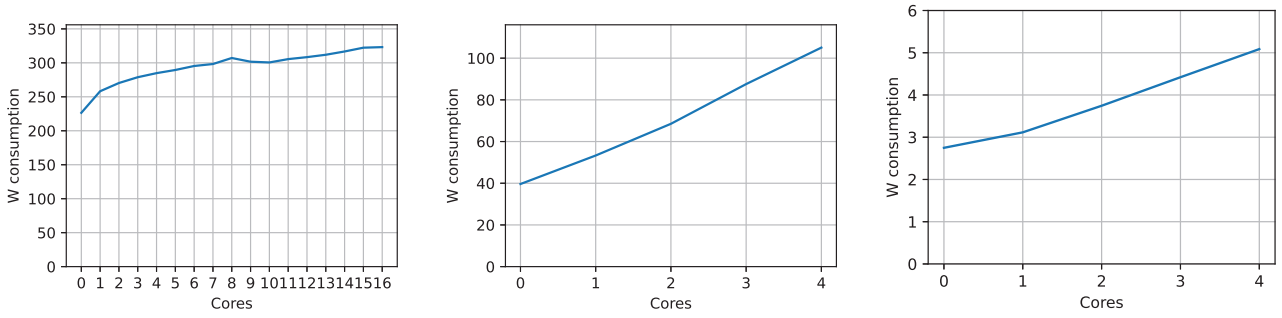


Fig. 3. The influence of the ensemble size in accuracy (red), energy efficiency (green), and throughput (blue) for each dataset executing the LBag algorithm on the Raspberry Pi.



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

Fig. 4. Energy consumption profile for each machine.

order to guarantee that the whole system achieved its steady state.

In this experiment we compare the baseline (Sequential) implementation of each algorithm, a parallel (multi-thread) implementation without mini-batching (B1), and three parallel versions with mini-batches of 50, 250, and 500 instances (B50, B250, and B500) respectively. Figures 5, 6, and 7 present the results from Raspberry Pi, i5, and Xeon 4208, respectively. The chart shows results for one dataset per row, whereas the columns show results for each algorithm. All charts in the same row have the same scale. The energy consumption (in Joules per instance - JPI) on the left Y-axis, whereas the average Delay (in milliseconds) per instance appears on the right Y-axis.

As a general remark, the energy efficiency of each algorithm varies according to its model complexity. For instance, all three versions of OzaBag show a better energy efficiency (i.e., smaller JPI) than the other algorithms because of the conservative resampling used in OzaBag (i.e., smaller λ). Such characteristic leads to slower-growing decision trees that are smaller, require fewer computations to train, and allow a faster traversal.

On the other hand, the algorithms with a more aggressive resampling (i.e., higher λ) presented a proportionally higher reduction in energy consumption (JPI) when compared to their best counterpart without mini-batching (o.e., baseline). Especially noteworthy is the case of SRP, which usually presents the highest energy consumption due to its deeper trees that require more training time. The mini-batching version generally presents a higher throughput, shortening the execution time while also having periods of low power consumption when sleeping and waiting for more instances.

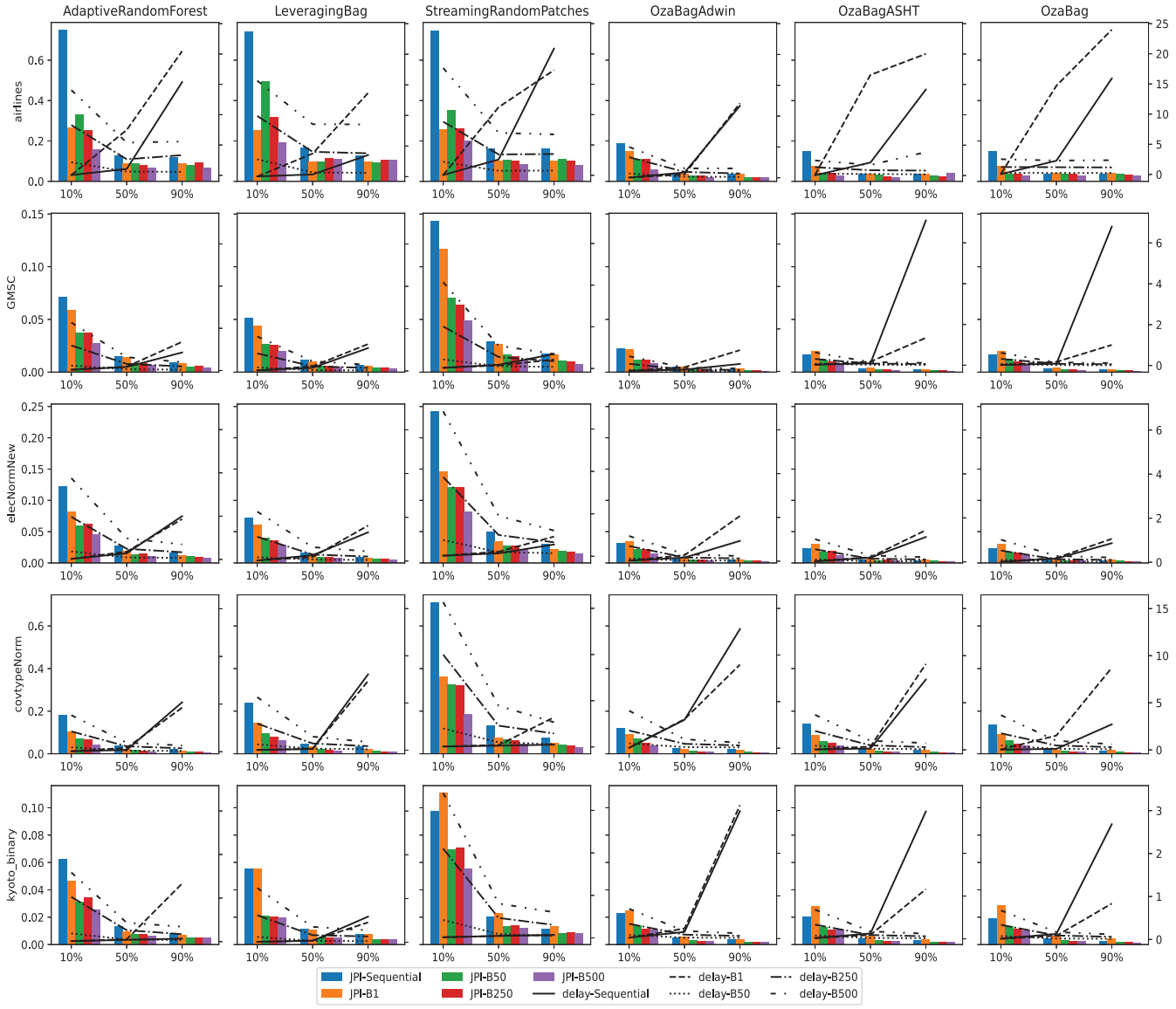


Fig. 5. Energy consumption and delay for the Raspberry Pi.

Overall, our experiments confirm that the main trade-offs discussed in [4] hold for all the machines and algorithms.

The energy efficiency gains can be explained as follows:

- Notice that the mini-batch is processed only in lines 6 and 11 of Algorithm 2, which are executed only when either the time elapsed exceeds the *timeout* or the mini-batch is full (i.e., it reaches L_{mb} instances). Otherwise, the loop (line 4) just appends the data instance in the mini-batch and waits (i.e., it enters in a sleep state) for the arrival of the next data instance. Because the thread sleeps for a while, it does not consume CPU cycles, so letting the DPM mechanisms implemented by the CPU hardware to turn off idle system components and thus save power;
- Even under high loads (i.e., sleep periods may tend to zero), mini-batching yields power reduction because it reduces cache misses (as demonstrated in [11]), thus accelerating the stream processing and reducing the number of cycles and memory accesses per instance.

Although mini-batching can improve energy efficiency, the delay resulting from extended and repeated sleep periods may hinder the idea of real-time processing, primarily when the rate of incoming instances is several times smaller than the mini-batch size. Such a phenomenon appears only in the B250 and B500 experiments. On the other hand, the B50 experiments present delays similar to the instance-by-instance implementations (Seq and B1) while having a better energy efficiency.

Regardless of the platform and dataset used, SRP has the worst energy efficiency across all the experiments. It happens because this algorithm produces deeper decision trees than the other complex algorithms (e.g., ARF and LBag), thus increasing the computational complexity for the stream processing.

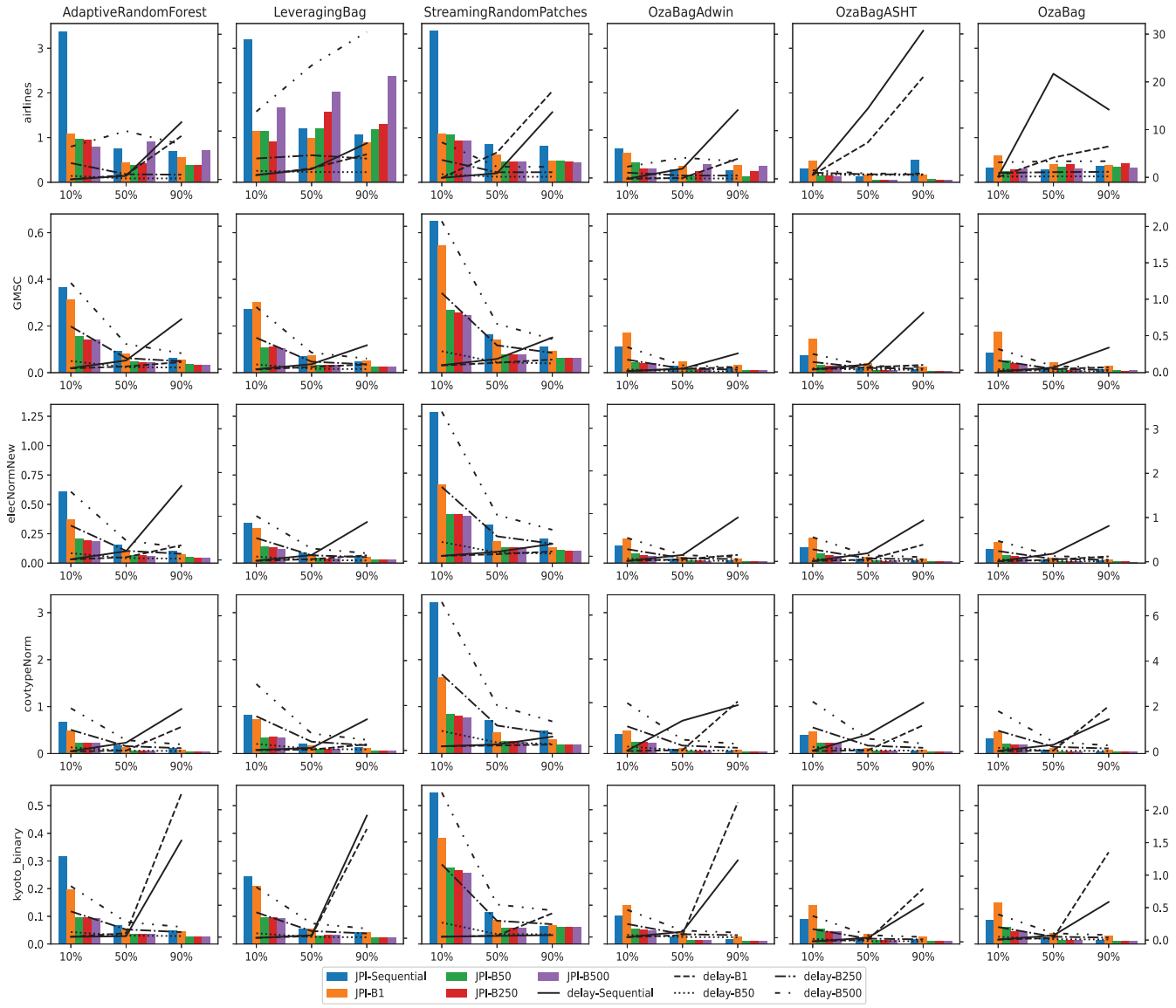


Fig. 6. Energy consumption and delay for the i5-2400.

The charts for the Raspberry Pi primarily present the expected behavior without significant deviations, which is not the case for the other two architectures. In Figure 6, some experiments show a higher delay with the larger mini-batches when compared to the baseline under the 90% workload. In these cases, the energy consumption also tends to be higher. Such behaviors can happen because the i5 has a faster clock and larger L1 cache, combined with smaller memory bandwidth. Together, these factors can raise a bottleneck in the communication channels since data is processed quickly after being stored in the L1 cache. In addition, a larger mini-batch can cause more idling periods where the processor is waiting to reach the desired mini-batch size, which unnecessarily extends the execution time.

In Figure 7, on the other hand, it is the opposite. Larger mini-batches usually have better efficiency, while the parallel version without mini-batches is worse than the baseline in many cases. Compared to the i5, this contrasting behavior is explained by the opposite characteristics of the Xeon 4208 processor, like a larger L3 cache, higher bandwidth, smaller L1 cache, and a slower clock.

Remarkably, mini-batches of 50 near instances yield better benefits in terms of delay reduction, energy efficiency, and smaller impact in predictive performance, while mini-batches larger than 50 instances presented diminishing returns. This result is consistent to our previous work in [11].

More detailed results on the energy efficiency gains are given in Tables III, IV, and V. They compare energy consumption between the best version without mini-batch (i.e., the best case between the orange and blue bars in the charts) and the mini-batch version. Negative values indicates the (percentage) reduction in energy consumption due to mini-batching. Cases where mini-batch consumed more energy have positive values and are in bold.

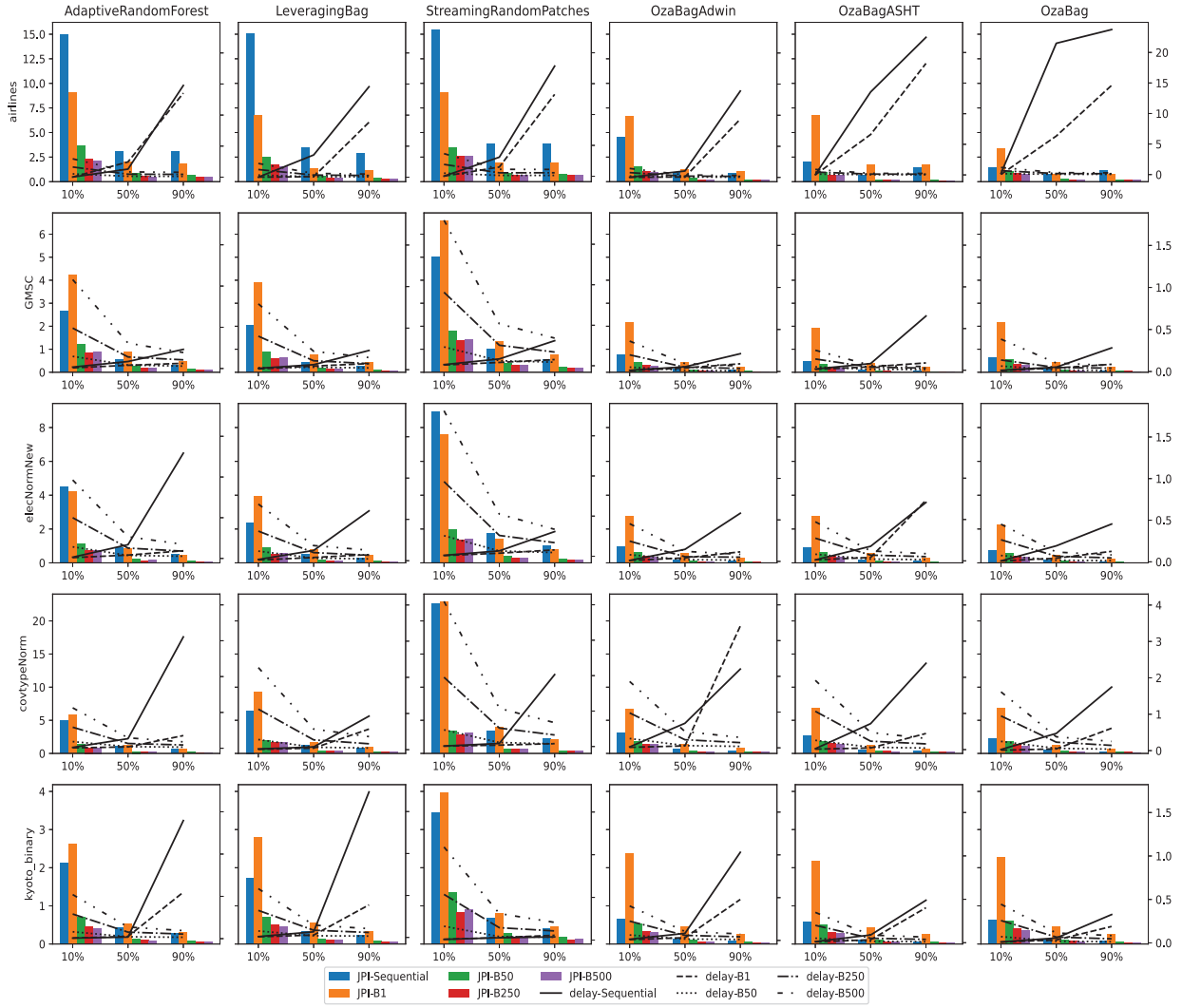


Fig. 7. Energy consumption and delay for the Xeon 4208.

TABLE III
PERCENTAGE DIFFERENCE BETWEEN THE BEST NON-MINI-BATCH VERSION AND THE MINI-BATCH VERSION ON THE RASPBERRY PI

	Airlines			GMSC			Electricity			Covertype			Kyoto		
Algorithm	10	50	90	10	50	90	10	50	90	10	50	90	10	50	90
ARF	-39.86	-24.87	-21.43	-53.42	-51.29	-47.81	-44.05	-42.69	-32.69	-58.19	-56.56	-52.42	-44.83	-35.61	-32.79
LBag	-25.04	**11.88**	**10.89**	-55.20	-51.70	-47.94	-53.48	-45.95	-39.18	-56.66	-55.94	-57.63	-64.64	-56.52	-48.65
SRP	-21.66	-18.04	-22.09	-57.89	-55.78	-51.54	-44.29	-37.97	-35.43	-48.95	-38.37	-36.49	-42.91	-42.20	-28.70
OBAd	-60.30	-51.74	-48.96	-61.54	-57.91	-52.18	-53.96	-50.63	-44.30	-55.02	-56.95	-64.73	-54.97	-53.28	-54.20
OBASHT	-62.30	-40.61	**17.97**	-55.46	-50.67	-58.96	-45.85	-40.91	-33.80	-56.95	-56.22	-60.48	-47.45	-45.95	-50.04
OB	-61.23	-27.51	-21.63	-55.14	-51.26	-58.31	-44.83	-39.42	-34.47	-57.53	-56.68	-65.04	-45.84	-44.35	-46.95

TABLE IV
PERCENTAGE DIFFERENCE BETWEEN THE BEST NON-MINI-BATCH VERSION AND THE MINI-BATCH VERSION ON THE I5-2400

	Airlines			GMSC			Electricity			Covertype			Kyoto		
Algorithm	10	50	90	10	50	90	10	50	90	10	50	90	10	50	90
ARF	-26.64	**112.66**	**27.88**	-55.09	-45.61	-39.58	-48.65	-40.06	-34.12	-55.05	-48.11	-42.40	-53.61	-34.79	-36.08
LBag	**47.50**	**105.85**	**164.66**	-62.05	-52.71	-48.87	-60.39	-50.86	-49.39	-54.99	-48.19	-43.84	-55.43	-38.61	-43.88
SRP	-14.89	-25.45	-8.34	-54.75	-44.62	-34.29	-40.39	-28.94	-21.54	-52.98	-42.79	-36.58	-33.10	-32.57	-3.14
OBAd	-54.64	**36.33**	**34.65**	-63.66	-57.72	-55.74	-58.95	-54.26	-52.44	-44.37	-39.91	-33.65	-50.78	-47.40	-37.44
OBASHT	-60.56	-58.60	-67.16	-63.21	-59.05	-51.85	-54.07	-47.84	-45.11	-40.51	-39.83	-35.71	-50.43	-46.33	-36.05
OB	-7.19	**5.56**	-13.34	-57.65	-54.94	-46.74	-51.92	-48.58	-45.28	-42.21	-40.24	-33.24	-47.55	-42.07	-30.06

It is possible to see in the tables that using mini-batch of 50 instances (MB50) improves both performance and energy efficiency in 259 out of 270 experiments (i.e., 96%). However, mini-batching has proven to increase the performance (e.g., throughput) in all the 270 experiments as illustrated in Figures 8, 9, and 10 and in Table VI.

TABLE V
PERCENTAGE DIFFERENCE BETWEEN THE BEST NON-MINI-BATCH VERSION AND THE MINI-BATCH VERSION ON THE XEON

	Airlines			GMSC			Electricity			Covertype			Kyoto		
Algorithm	10	50	90	10	50	90	10	50	90	10	50	90	10	50	90
ARF	-76.81	-75.08	-75.30	-67.09	-66.98	-67.38	-81.74	-80.47	-80.30	-82.64	-83.06	-82.33	-80.12	-77.36	-80.25
LBag	-76.71	-73.22	-74.66	-68.16	-68.30	-68.92	-76.63	-76.37	-77.18	-74.41	-71.04	-71.24	-72.75	-70.85	-72.61
SRP	-71.57	-66.02	-67.11	-71.23	-67.91	-67.84	-81.06	-78.40	-76.87	-86.37	-81.20	-82.01	-73.70	-71.24	-68.50
OBAd	-78.07	-77.87	-80.59	-62.25	-63.51	-63.70	-63.50	-64.24	-64.56	-56.87	-55.37	-55.34	-52.19	-54.16	-51.48
OBASHT	-69.59	-79.84	-90.91	-60.48	-61.67	-62.04	-59.17	-58.30	-61.89	-48.61	-49.38	-48.26	-52.69	-53.35	-52.64
OB	-44.22	-77.02	-81.20	-51.25	-52.18	-52.74	-53.49	-54.06	-57.95	-50.19	-48.38	-49.19	-44.46	-45.90	-43.83

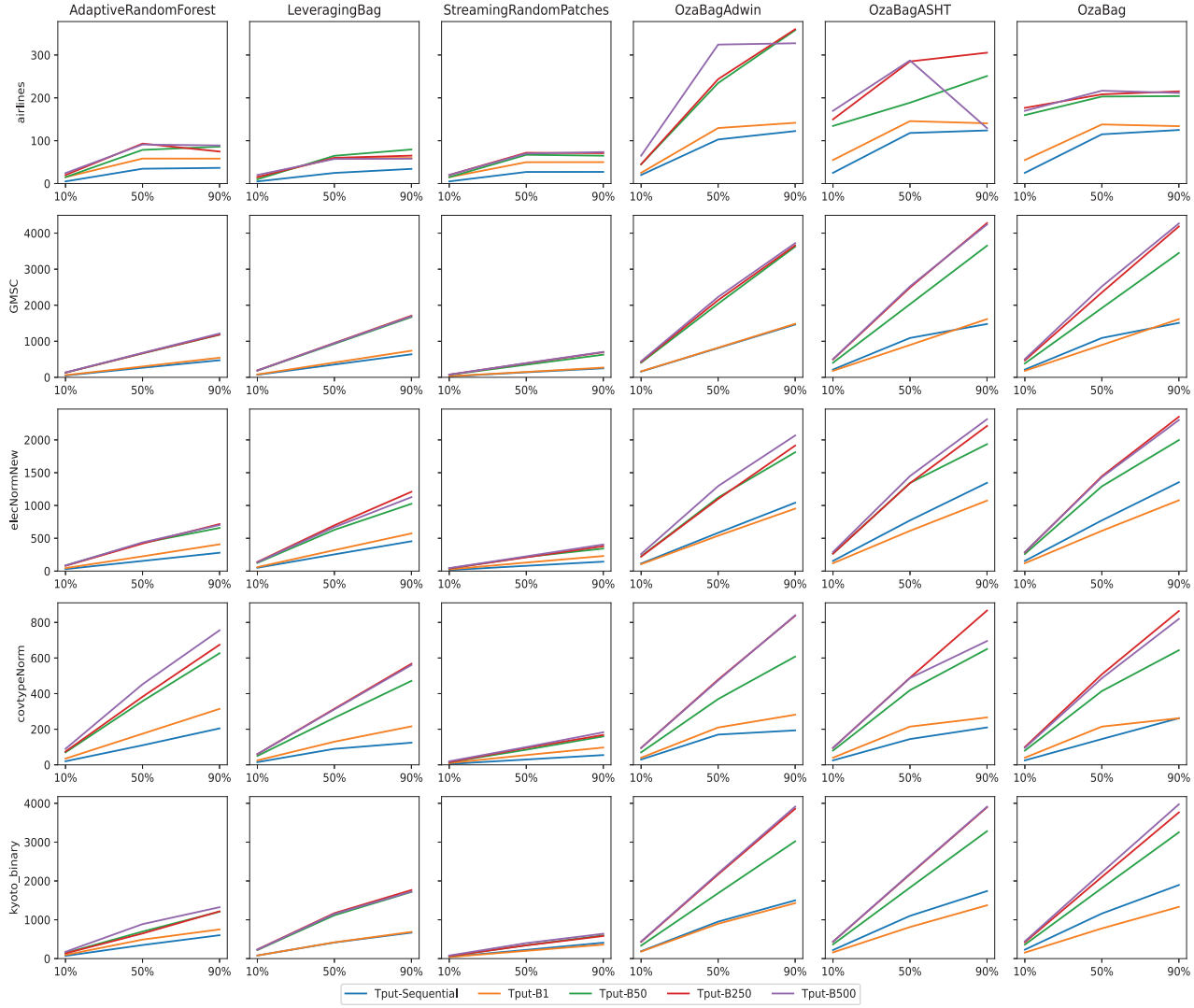


Fig. 8. Throughput for the Raspberry Pi

TABLE VI
THROUGHPUT FOR THE ALGORITHM LEVERAGINGBAG USING THE AIRLINES DATASET

Algorithm	10%	50%	90%
Sequential	9.99	37.05	46.56
B1	34.96	45.68	53.46
B50	34.92	74.44	78.12
B250	49.19	58.95	71.05
B500	20.01	20.80	16.21

In summary, mini-batching can support consistent improvements in energy consumption and time performance across the experiments. First, mini-batching improves the RD, reducing the number of CPU cycles to process each data instance and spending less energy. Second, the algorithm creates sleeping periods while accumulating new instances to compose the mini-batch, which allows the DPM strategies to turn off idle subsystems of the processor to save energy.

Based on our results, we can say that reducing energy consumption is related to reducing the Reuse Distance (RD) metric. This relation exists because the mini-batching strategy applies a division by a constant number to the RD metric, defined in [11] as:

$$RD = \left\lceil \frac{nm^2}{b} \right\rceil,$$

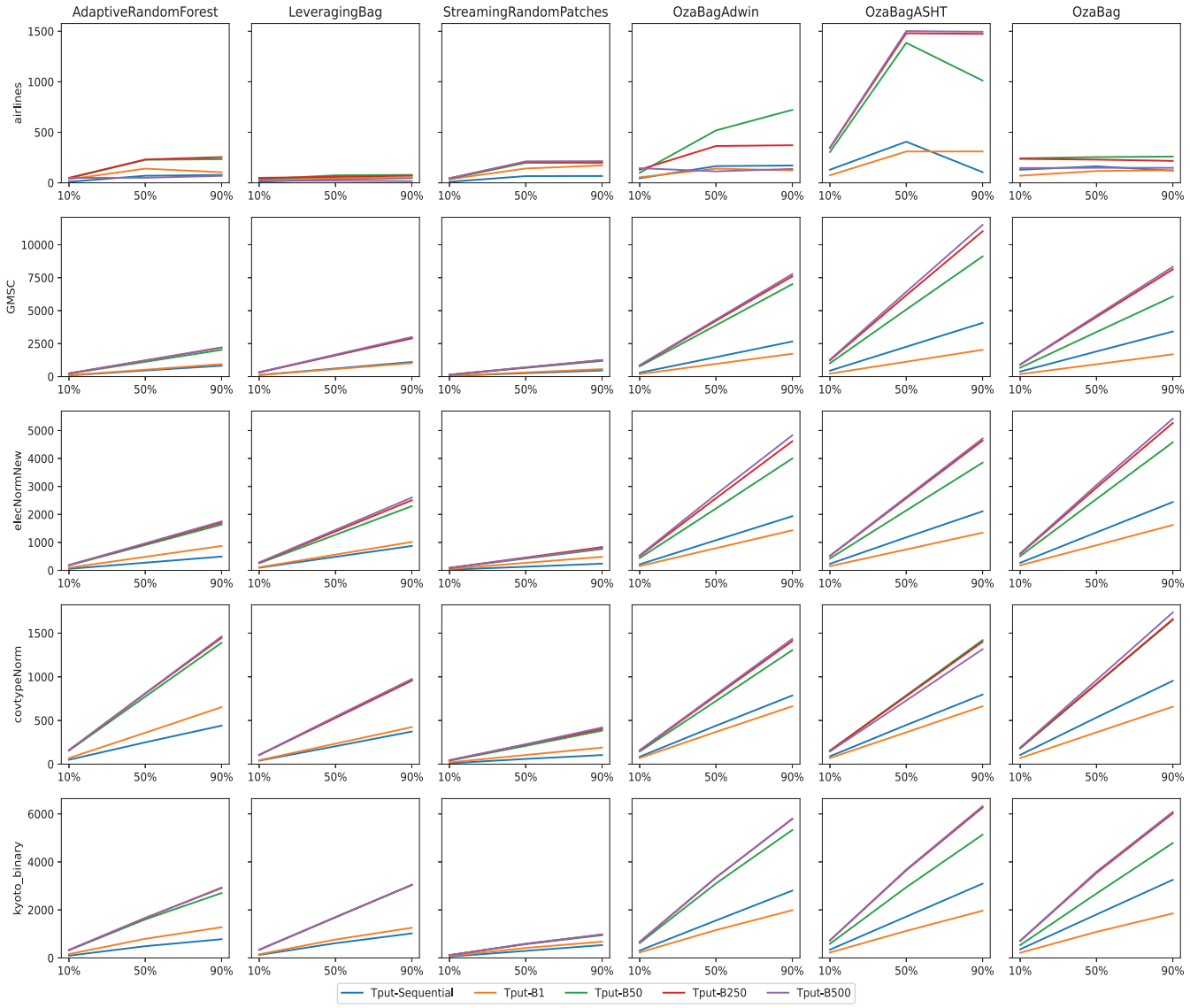


Fig. 9. Throughput for the i5-2400

where n is the number of instances processed, m is the ensemble size, and b is the mini-batch size.

The b constant in the equation means that the marginal gains from increasing the mini-batch size will eventually become negligible. In addition, if we consider an infinite stream of data, we can discard n from the equation.

When n is infinite, RD becomes a function of the ensemble size and mini-batch size, causing the marginal gains from b to become negligible even faster. In this case, RD can be defined as an "instant" Reuse Distance (*iRD*):

$$iRD = \left\lceil \frac{m^2}{b} \right\rceil. \tag{7}$$

By taking the energy model from [9] as a base and using the *iRD* (i.e., equation (7)), it is possible to extend the model to account for the mini-batching by the following equation:

$$Energy\_consumption\_MB = (SC * iRD),$$

where *SC* refers to the sequential consumption defined in [9]. Note that a single classifier has an ensemble_size equal to 1 and a mini-batch size equal to 1, therefore keeping the model correct.

Also, notice that mini-batching causes trade-offs between energy consumption, time metrics (delay, and throughput), and accuracy. One trade-off favours throughput and energy efficiency in exchange for accuracy. While we can noticeably increase the energy efficiency and throughput of the algorithm by reducing the ensemble size, we get only a negligible deterioration in the accuracy.

Another trade-off emerges regarding the average delay in processing the instances and the energy consumption. We observed more significant improvements in time performance and energy savings by using mini-batching at low loads, with the downside of incurring a higher average delay. Conversely, when the workload is high, mini-batching increases throughput. In addition, the energy reduction is present but not as intense, and the delay is smaller than the baseline. Such results

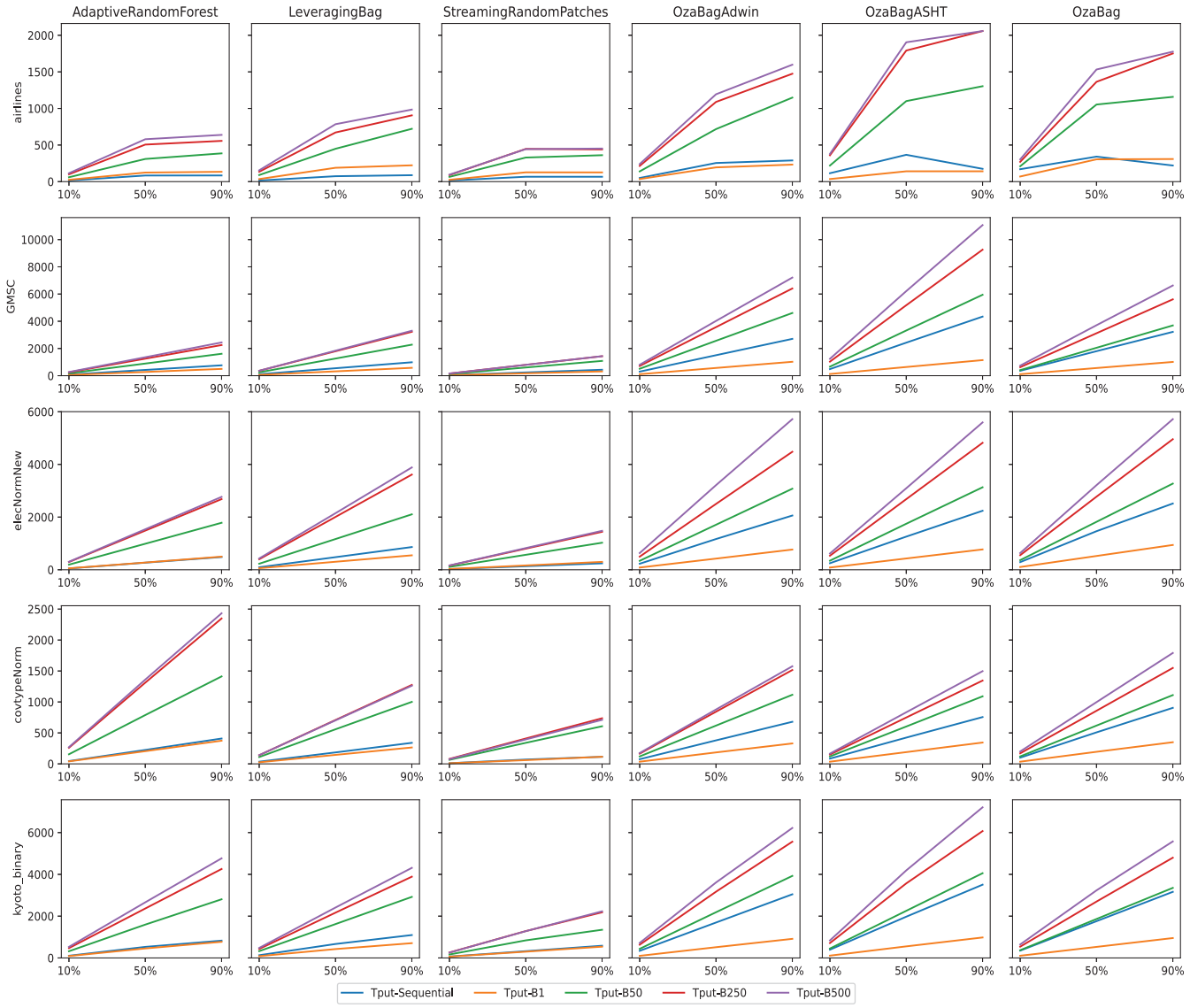


Fig. 10. Throughput for the Xeon 4208

align with those found in the literature, where the least complex algorithm is the best when evaluating energy consumption alone.

Moreover, we present the comparison of the trade-offs found in our results concerning the four categories presented in [4]. The categories are time cost, energy consumption, predictive performance, and memory cost.

The mini-batching can reduce both the induction and inference time regarding the time cost. However, deferring the testing to the end of the mini-batch can affect predictive performance.

Regarding energy consumption, mini-batching can reduce energy consumption in two ways. (*i*) by reducing the number of processor cycles to execute both the training and testing operations, it can minimize the number of cache misses. (*ii*) the idleness periods created when building the mini-batches trigger the DMP mechanisms more frequently, thus saving more power. Again, the deferring of the testing to the end of the mini-batch affects predictive performance.

Usually, the predictive performance can be improved at the cost of computational resources. Such a relationship holds for mini-batching, where reducing the mini-batch size increases the computational cost and predictive performance. Conversely, increasing the mini-batch size reduces the time cost and predictive performance.

Regarding memory cost, mini-batching increases the algorithm's memory footprint for storing the instances and the votes of the ensemble. However, this increase is negligible compared to the main benefit mini-batching provides: significantly improving memory access locality by minimizing the number of cache misses. At this point, we make two observations. First, the related work analyzes the memory footprint of the algorithms. However, we claim that the memory access pattern of the algorithms is more impacting than the memory footprint on the time and energy costs. Second, a possible insight points to the need to study new data structures (e.g., which extend the classical VFDT) that can better fit the cache hierarchy of the processors.

Despite the trade-offs observed, it is possible to balance them to avoid significant loss in predictive performance. Generally, the marginal benefits from increasing the mini-batch size past 50 instances become smaller and smaller across all workloads, algorithms, datasets, and platforms tested. On the other hand, under the light workloads, the delay always increases as we increase the mini-batch size. Therefore, both results corroborate past results in [11], which suggests that 50 instances is a good choice for balancing predictive performance and execution performance improvement. This might be a good guess when there is no further knowledge of the application. Different combinations of algorithm, dataset, platform, and workload will have different optimal values.

VII. Conclusion

Ensemble learning is a fruitful approach to improve the performance of ML models by combining several single models. Ensembles are also popular in a data stream processing context, where they achieve remarkable predictive performance. Examples of this class include algorithms such as Adaptive Random Forest, Leveraging Bag, and OzaBag. Despite their relevance, many aspects of their efficient implementation remained to be studied after their original proposals. For example, the original Adaptive Random Forest implementation included a simple multi-thread version, but it did not consider energy efficiency or mini-batches to improve the overall run time.

Previous literature that analyzed the energy efficiency did not consider bagging ensembles [4] or performed a slightly different analysis, not evaluating time measures [9]. To bridge this gap, in this paper, we use an experimental testbed to evaluate six state-of-art bagging algorithms (OzaBag, OzaBag Adaptive Size Hoeffding Tree, Online Bagging ADWIN, Leveraging Bagging, Adaptive RandomForest, and Streaming Random Patches) with different complexity to illustrate several use cases where task-parallelism can be stressed. The scenario implemented in the testbed sends data through the network at different load intensities using five widely popular machine learning benchmark datasets with varied characteristics. Data is processed on three computer platforms to show that the mini-batching strategy can improve the ensembles used to classify data streams independently of the hardware. We show that the energy consumption is reduced in 96% of the tested cases, indicating that it works for all bagging algorithms.

Also, notice that mini-batching causes a trade-off between energy consumption, time metrics (delay and throughput), and accuracy. While the mini-batching increases the throughput, delay, and energy efficiency, larger mini-batches can modify the accuracy of the algorithms. Previous studies on mini-batching performance corroborate this result [11]. However, as demonstrated in our experiments, it is possible to balance this trade-off. The mini-batch size of 50 instances is a good balance demonstrated for practically all the experimental scenarios studied. Finally, we presented energy consumption models for bagging ensembles, which extend the energy model from the literature (proposed in [9]) to estimate the final energy consumption of bagging ensembles with mini-batching.

In future work, we intend to investigate if it is possible to improve the solution by using an adaptive mini-batching size regarding predictive performance, throughput, delay, and energy consumption.

References

[1] W. Z. Khan, E. Ahmed, S. Hakak, I. Yaqoob, and A. Ahmed, "Edge computing: A survey," *Future Gener. Comput. Syst.*, vol. 97, pp. 219–235, Aug. 2019.

[2] A. Shakarami, M. Ghobaei-Arani, and A. Shahidinejad, "A survey on the computation offloading approaches in mobile edge computing: A machine learning-based perspective," *Comput. Netw.*, vol. 182, Dec. 2020, Art. no. 107496.

[3] X. Yu, C. Cecati, T. Dillon, and M. G. Simoes, "The new frontier of smart grids," *IEEE Ind. Electron. Mag.*, vol. 5, no. 3, pp. 49–63, 2011.

[4] J. F. Lopes, E. J. Santana, V. G. T. da Costa, B. B. Zarpelão, and S. Barbon, "Evaluating the four-way performance trade-off for data stream classification in edge computing," *IEEE Trans. Netw. Service Manag.*, vol. 17, no. 2, pp. 1013–1025, Jun. 2020.

[5] D. A. Coutinho et al., "Performance and energy trade-offs for parallel applications on heterogeneous multi-processing systems," *Energies*, vol. 13, no. 9, p. 2409, May 2020.

[6] X. Jin, L. Li, F. Dang, X. Chen, and Y. Liu, "A survey on edge computing for wearable technology," *Digit. Signal Process.*, vol. 125, Jun. 2022, Art. no. 103146. [Online]. Available: https://doi.org/10.1016/j.dsp.2021.103146

[7] J. Gama, I. Žliobaitė, A. Bifet, M. Pechenizkiy, and A. Bouchachia, "A survey on concept drift adaptation," *ACM Comput. Surveys* , vol. 46, no. 4, pp. 1–37, 2014.

[8] H. M. Gomes, J. P. Barddal, F. Enembreck, and A. Bifet, "A survey on ensemble learning for data stream classification," *ACM Comput. Surveys*, vol. 50, no. 2, pp. 1–36, 2017.

[9] E. García-Martín, A. Bifet, and N. Lavesson, "Energy modeling of Hoeffding tree ensembles," *Intell. Data Anal.*, vol. 25, no. 1, pp. 81–104, 2021.

[10] G. Cassales, H. Gomes, A. Bifet, B. Pfahringer, and H. Senger, "Improving parallel performance of ensemble learners for streaming data through data locality with mini-batching," in *Proc. IEEE Int. Conf. High Perform. Comput. Commun. (HPCC)*, 2020, pp. 138–146.

[11] G. Cassales, H. Gomes, A. Bifet, B. Pfahringer, and H. Senger, "Improving the performance of bagging ensembles for data streams through mini-batching," *Inf. Sci.*, vol. 580, pp. 260–282, Nov. 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0020025521008938

[12] A. C. Orgerie, M. D. De Assuncao, and L. Lefevre, "A survey on techniques for improving the energy efficiency of large-scale distributed systems," vol. 46, no. 4, p. 47, 2014. [Online]. Available: http://dx.doi.org/10.1145/2532637

[13] L. A. Barroso and U. Hölzle, "The case for energy-proportional computing," *Computer*, vol. 40, no. 12, pp. 33–37, Dec. 2007.

[14] L. Benini, A. Bogliolo, and G. De Micheli, "A survey of design techniques for system-level dynamic power management," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 8, no. 3, pp. 299–316, Jun. 2000.

[15] D. C. Snowdon, S. Ruocco, and G. Heiser, "Power management and dynamic voltage scaling: Myths and facts," in *Proc. Workshop Power Aware Real-Time Comput.*, vol. 31, 2005, p. 34.

[16] J. Vieira, R. P. Duarte, and H. C. Neto, "kNN-STUFF: KNN Streaming unit for Fpgas," *IEEE Access*, vol. 7, pp. 170864–170877, 2019.

[17] E. G. Martin, N. Lavesson, and H. Grahn, "Energy efficiency in data stream mining," in *Proc. IEEE/ACM Int. Conf. Adv. Social Netw. Anal. Min. (ASONAM)*, 2015, pp. 1125–1132.

[18] I. Amezzane, A. Berrazzouk, Y. Fakhri, M. E. Aroussi, and M. Bakhouya, "Energy consumption of batch and online data stream learning models for smartphone-based human activity recognition," in *Proc. 4th World Conf. Complex Syst. (WCCS)*, 2019, pp. 1–5.

[19] V. G. T. da Costa, A. C. P. de Leon Ferreira de Carvalho, and S. Barbon, "Strict very fast decision tree: A memory conservative algorithm for data stream mining," *Pattern Recognit. Lett.*, vol. 116, pp. 22–28, Dec. 2018.

[20] V. G. Turrisi da Costa, E. J. Santana, J. F. Lopes, and S. Barbon, "Evaluating the four-way performance trade-off for stream classification," in *Proc. Int. Conf. Green, Pervasive, Cloud Comput.*, 2019, pp. 3–17.

[21] J. A. Silva, E. R. Faria, R. C. Barros, E. R. Hruschka, A. C. D. Carvalho, and J. Gama, "Data stream clustering: A survey," *ACM Comput. Surveys*, vol. 46, no. 1, pp. 1–31, 2013.
[22] P. Domingos and G. Hulten, "Mining high-speed data streams," in *Proc. 6th ACM SIGKDD Int. Conf. Knowl. Disc. Data Min.*, Sep. 2000, pp. 71–80.
[23] L. Breiman, "Bagging predictors," *Mach. Learn.*, vol. 24, no. 2, pp. 123–140, 1996. [Online]. Available: https://doi.org/10.1007/BF00058655
[24] Y. Freund and R. E. Schapire, "Experiments with a new boosting algorithm," in *Proc. ICML*, 1996, pp. 148–156.
[25] N. C. Oza and S. Russell, "Online bagging and boosting," in *Proc. 8th Int. Workshop Artif. Intell. Statist.*, Jan. 2001, pp. 105–112.
[26] A. Bifet, G. Holmes, B. Pfahringer, R. Kirkby, and R. Gavaldà, "New ensemble methods for evolving data streams," in *Proc. 15th ACM SIGKDD Int. Conf. Knowl. Disc. Data Min.*, 2009, pp. 139–148.
[27] A. Bifet, G. Holmes, and B. Pfahringer, "Leveraging bagging for evolving data streams," in *Machine Learning and Knowledge Discovery in Databases*, J. L. Balcázar, F. Bonchi, A. Gionis, and M. Sebag, Eds. Berlin, Germany: Springer, 2010, pp. 135–150.
[28] H. M. Gomes et al., "Adaptive random forests for evolving data stream classification," *Mach. Learn.*, vol. 106, no. 9, pp. 1469–1495, 2017. [Online]. Available: https://doi.org/10.1007/s10994-017-5642-8
[29] C. Manapragada, G. I. Webb, and M. Salehi, "Extremely fast decision tree," in *Proc. 24th ACM SIGKDD Int. Conf. Knowl. Disc. Data Min.*, 2018, pp. 1953–1962.
[30] A. Bifet and R. Gavaldà, "Learning from time-changing data with adaptive windowing," in *Proc. 7th SIAM Int. Conf. Data Mining*, vol. 7, Apr. 2007, pp. 1–17.
[31] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, 2001.
[32] H. M. Gomes, J. Read, and A. Bifet, "Streaming random patches for evolving data stream classification," in *Proc. IEEE Int. Conf. Data Min. (ICDM)*, 2019, pp. 240–249.
[33] A. Bifet et al., "MOA: Massive online analysis, a framework for stream classification and clustering," in *Proc. 1st Workshop Appl. Pattern Anal.*, 2010, pp. 44–50.
[34] L. Yuan, C. Ding, W. Smith, P. Denning, and Y. Zhang, "A relational theory of locality," *ACM Trans. Archit. Code Optim.*, vol. 16, no. 3, pp. 1–26, 2019.
[35] R. Pereira et al., "Energy efficiency across programming languages: How do energy, time, and memory relate?" in *Proc. 10th ACM SIGPLAN Int. Conf. Softw. Lang. Eng.*, 2017, pp. 256–267. [Online]. Available: https://doi.org/10.1145/3136014.3136031

Guilherme Cassales is a Postdoctoral Fellow with the AI Institute, University of Waikato, New Zealand. His research interests include HPC and data stream mining.

Heitor Murilo Gomes is an Assistant Professor (Lecturer) with the Victoria University of Wellington, New Zealand. His main research area is machine learning for data streams, specifically concept drift detection, ensemble methods and semi-supervised learning algorithms. He contributes to open data stream mining projects, such as the Massive Online Analysis framework.

Albert Bifet (Member, IEEE) designs and builds applications on machine learning for data streams, at the intersection of data science, algorithms, and business. His research mission is to develop new Artificial Intelligence algorithms and tools to gain insights from Big Data using Machine Learning for Big Data streams and large-scale data analytics. He manages international open-source projects, providing leading strategy, and technical approaches, to deliver high-quality software and academic research.

Bernhard Pfahringer is an Experimental Machine Learning Researcher. He enjoys building ML systems, and to measure and evaluate them on both artificial and real world data. The goal is to gain a better understanding of and insights into their inner workings. This in turn supports improvements for existing algorithms, as well as the development of genuinely novel algorithms.

Hermes Senger is an Enthusiastic Researcher for improving the efficiency of applications across different computer architectures, from tiny devices with limited capabilities to supercomputers and distributed systems. The research goals may vary, from improving existing algorithms or developing new ones to enhance the efficiency of applications on either existing or new architectures. He is currently an Associate Professor with the Federal University of São Carlos, Brazil.