Assessing Hardware and Software Approaches for Energy Efficiency of Data Stream Processing on the Edge

Reginaldo Luna*, Guilherme Cassales[†], Heitor Murilo Gomes[‡], Bernhard Pfahringer[†], Albert Bifet[†], Hermes Senger*
*Universidade Federal de São Carlos, São Carlos, Brazil
[†]University of Waikato, Hamilton, New Zealand
[‡]Victoria University of Wellington, Wellington, New Zealand

***Abstract*—Edge Computing (EC) has emerged as a solution to reduce energy demand and greenhouse gas emissions from digital technologies. EC supports low latency, mobility and location awareness for delay-sensitive applications by bridging the gap between cloud computing services and end-users. Machine learning (ML) methods have been applied in EC for data classification and information processing. Ensembles learners have often proven to yield high predictive performance on data stream classification problems. This study evaluates four strategies for improving energy efficiency in data stream classification using six state-of-the-art ensemble algorithms and four benchmark datasets. Results show that software strategies can significantly reduce energy consumption in 95% of the experiments. Mini-batching improved energy efficiency by 96% on average and 169% in the best case. Likewise, mini-batching with loop fusion improved energy efficiency by 136% on average and 456% in the best case. These strategies also support better control of the balance between performance, energy efficiency and accuracy.**

***Index Terms*—energy efficiency, ensembles, data stream, parallelization.**

I. Introduction

Edge Computing (EC) is a paradigm that moves the services and utilities of cloud computing closer to data sources (e.g., sensors), and closer to the end-user, providing low latency, location awareness, and better efficiency for delay-sensitive mobile applications [1]. EC is an enabling technology for developing applications in Internet of Things (IoT), 5G, online gaming, augmented reality (AR), smart grids, and real-time video analytics, among others. In such applications, a large number of sensors can be deployed to continuously collect data from the surrounding environment over medium to large geographical areas [2]. Because most sensors are resource-constrained, they cannot run expensive or data-intensive tasks like machine learning (ML) algorithms. It is more efficient to process data on resources nearby (in the edge of the network) with low latency [3], [4]. Often, such tasks can execute on energy- and resource-constrained mobile devices or low-end servers placed nearby [5]. In these scenarios, we see increasing adoption of ML techniques to execute tasks like data classification, anomaly detection, network intrusion detection for cybersecurity, real-time image classification and segmentation, driving support applications, autonomous driving, and others. However, executing ML algorithms on mobile devices or edge servers while coping with dynamic environments where data streams are potentially infinite and have non-stationary behavior, creates challenges for efficient energy consumption.

Historically, ML research focused on improving predictive performance without considering computational resources or energy consumption constraints. On the algorithmic side, the ML field is shifting towards processing continuous data streams to face the challenges above, where requirements like single pass, response time, and constant memory usage are imposed [6]. Ensembles of classifiers are ML techniques that demonstrated noticeable accuracy for classifying data streams [7] by using several weak learners which can produce accurate results. Nevertheless, little effort has been made to reduce the energy consumption of data stream algorithms [8].

Aiming to fill this gap, *mini-batching* was proposed as a technique for optimizing the efficiency of ensembles for processing data streams [9]. Mini-batching provides optimal memory locality for data stream ensembles [10]. The study in [11] demonstrates the benefit of mini-batching to reduce energy consumption for the classification of data streams with bagging ensembles. Despite the benefits, mini-batching presented two main limitations. The technique has been applied only for the classification or the training phase (the most computationally intensive one), which hinders its cache reuse between these two phases of the algorithm [10] and reduces its change detection capacity [11]. In the present work, we tackle these limitations by changing the way mini-batching is applied to the continuous training and test loops for the classification of data streams. In this paper, we provide the following contributions:

1) Presenting a novel study that compares mini-batching and hardware strategies for improving the performance and energy efficiency of bagging ensembles. Our results show that mini-batching strategies lead to better performance in 96% of the evaluated cases;
2) Enhancing the mini-batching approaches from the liter-

ature [9], [10] by fusing the classification and training steps, improving the performance, energy efficiency, and predictive performance for bagging ensembles;
3) Evaluating the trade-offs between time performance, energy efficiency, and predictive performance of the four strategies for classifying online data streams.

This article is organized as follows: Section II discusses the related works. Section III presents the ensemble algorithms evaluated in our experiments. We present our new mini-batching strategy for bagging ensembles in Section IV, and the experimental evaluation in Section V. We present our conclusion in Section VI.

II. Related Works

Various strategies have been proposed to save energy [12]. Dynamic power management (DPM) strategies can selectively turn off (or reduce the performance of) system components when they are idle (or partially unexploited), aiming to reduce energy consumption [13]. Dynamic voltage and frequency scaling (DVFS) automatically reduce the clock frequency of processing cores aiming to save power [14]. The work in [15] combines frequency scaling and core limiting to optimise performance and energy savings using the Pareto frontier. Although effective, the strategy affects all the applications executing on the same machine. The proposal in [16] implements the k-nearest neighbors (kNN) algorithm on FPGA (Field-Programmable Gate Array) hardware. Despite accelerating the execution by 60x, and improving energy efficiency by 50x, the solution is both algorithm- and hardware-specific.

Using software approaches, [17] demonstrated that energy consumption can be reduced by up to 92.5% while maintaining accuracy when varying the parameters of the Very Fast Decision Tree (VFDT) algorithm. The work in [18] proposed an extension of the VFDT which reduces memory usage and execution time by minimizing tree growth while maintaining competitive predictive performance. In [19], authors discuss performance trade-offs and conclude that the most complex method delivers the best predictive performance at the expense of worse memory and energy performance. The work in [8] proposed to reduce the energy consumption of Hoeffding Trees [1] ensembles by adapting the minimum number of instances required for a node split. This method reduces energy consumption by 21% with only a small impact on accuracy. Although such works propose energy-efficient techniques, they benefit a limited set of specific algorithms.

The *mini-batching* [9], [10] approach was proposed to improve the performance of bagging ensembles for data stream classification. Later in [11], the authors proposed a strategy that defers the processing *mini-batching* to introduce short periods of CPU idleness that can trigger DPM techniques to reduce these algorithms' power consumption and processing latency. However, mini-batching has been applied only for either the classification or the training phase (only the most computationally intensive one). Thus, there is no cache reuse between these two phases of the algorithm [10]. Furthermore, when mini-batching is applied only to the classification phase, the training is postponed until the end of the processing of a mini-batch, which reduces the change detection capacity of the algorithms [11].

In the present work, we tackle these limitations by changing the way mini-batching is applied to the continuous training and test loops for the classification of data streams. To this end, we introduce a new *mini-batching* strategy that fuses the training and testing loops. We evaluated the new strategy on six state-of-the-art bagging ensembles for the classification of data streams, and compare it with the strategy proposed in [11]. We also compare the mini-batching approaches with two hardware strategies supported by modern commodity CPUs (DPM, and VFS).

III. 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 $\rightarrow \infty$) [21]. 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, the Hoeffding Tree (HT) [20] is considered one of the most widely used algorithms. It is specifically designed to handle massive data streams by adapting incrementally. However, despite its capability to make splits with limited data, the HT has limitations when it comes to modeling complex learning problems using a single tree. To overcome this issue, a popular strategy is to ensemble several models using Bagging [22]. Although being proposed over 25 years ago, Bagging remains an effective method for reducing error without resorting to complex models, such as deep neural networks, that are not trivial to train and fine-tune. By utilizing the Poisson distribution, Bagging creates several different subsets of the data to repeatedly train with some instances while ignoring others, enhancing its efficacy. Next, we briefly describe six ensemble algorithms that evolved from the original Bagging to an online (streaming) setting by Oza and Russell [23]. Despite other decision tree algorithms [24], the HT algorithm is often chosen as the base model for the online bagging algorithms. It may not be the most accurate individually, but it does yield good predictive performance without requiring too many computational resources.

Online Bagging (OzaBag - OB) [23] 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, 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 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

[1] Hoeffding Tree [20] and VFDT are often used as synonyms in the literature.

detrimental to an online learning setting [7]. We chose this algorithm because it can be considered a baseline for bagging ensembles. Furthermore, OB does not include any additional technique to improve predictive performance, such as drift detectors or recovery strategies. Therefore, it is an efficient ensemble method, although its predictive performance is lower compared to more advanced ensemble techniques.

OzaBag Adaptive Size Hoeffding Tree (OBagASHT) [25] combines 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). OBagASHT was designed to improve the predictive performance by enforcing the creation of diverse trees. Effectively, diversity is enforced 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. This algorithm creates tasks with heterogeneous workloads. For example, when scheduling 100 parallel tasks on a multi-core system, the difference in tree size (and in the 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) [25] combines OzaBag with the ADAptive WINdow (ADWIN) [26] 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 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 [28] to the streaming setting. A 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 [20] 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 signaled, another model (a 'background tree') is created and trained without affecting the ensemble predictions. If the warning escalates to a drift signal, the associated tree is replaced by its background tree. Although the background tree incurs additional storage requirements, its co-existence with the foreground tree is often short-lived [29]. In addition, such a mechanism allows the ensemble to have a faster drift recovery. The reason is that new models will have seen a few instances and started learning the new data distribution, which generates a more homogeneous load between the parallel tasks.

Streaming Random Patches (SRP) [30] 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 do not modify the base learners behaviour. This happens because ARF's strategy performs the random subset selection at the learning node level of a tree (i.e., local randomization), whereas SRP randomly selects the subset of features for a whole tree (i.e., global randomization). Still, in [30] the experiments were made using Hoeffding trees and showed that SRP can produce deeper trees, which may lead to increased diversity in the ensemble. The deeper and fast growing trees increases the computational cost (i.e., runtime) rapidly, creating the heaviest workload of the algorithms.

IV. Power Saving Strategies

One of the most widely used algorithms for processing data streams, the Hoeffding Tree (HT) is designed to handle massive data streams by adapting incrementally. However, split operations demand memory allocation/release to update machine learning models (i.e., data structures) during the continuous learning/training process of online data streams [20]. Furthermore, the HT data structures are frequently larger than the cache memories, thus raising the number of cache misses. Mini-batching is a strategy proposed to reduce cache

misses and improve the performance of bagging ensembles [10].

A. Mini-batching

In essence, batching techniques consist of grouping instances of a dataset and processing them together. The technique can be applied in different contexts, for different purposes. Mini-batching is commonly used in deep learning, where the training data is divided into smaller subsets called mini-batches. By using batches instead of processing one data instance at a time, the training process becomes more efficient and memory-friendly. In Spark, data instances are grouped into small batches for efficient fault recovery [31]. However, the mini-batching studied in this paper differs from the previous scenarios.

The mini-batching here refers to a specific optimization for improving the data locality of bagging ensembles, as defined in [10]. The mini-batching strategy groups several data instances of a stream into small chunks (named mini-batches) to be processed by each learner of the ensemble. Each learner is executed by a parallel task that iterates over all the instances within a mini-batch. When a learner is invoked, its data structures are loaded into the caches to process the first data instance of the mini-batch, and reused for the processing of the next instances (different from the case of processing a single instance at a time like in the naive algorithm). Thus, mini-batching reduces cache misses and improves performance. As proved in [10], mini-batching can significantly reduce the *reuse distance* metric, which reduces the number of cache misses and the number of processor cycles, thus reducing the execution time and the energy consumed to process the data stream [11].

Algorithm 1 Mini-batching for data streams

```
1:  Input: ensemble E, data stream S, batch size limit L_mb
2:  T ← E.getCollectionOfTrainers()
3:  B ← new MiniBatch()
4:  while there are instances arriving in stream S do
5:      if ElapsedTime > Timeout OR B.size = L_mb then
6:          E.process_minibatch(B)
7:      end if
8:      I = S.getNextInstance()              ▷ Blocking wait
9:      B.append(I)
10: end while
11: E.process_minibatch(B)
```

Mini-batching is described in Algorithm 1. The loop starting in line 4 iterates over the instances of the data stream, accumulating instances (in lines 8 and 9). The processing of the mini-batch is deferred until a timeout is reached or the mini-batch is full (line 5). The mini-batch size (L_{mb}) and timeout value in line 5 can be configured as parameters of the algorithm. The blocking wait for the next arriving instances is important in two ways: (i) it releases CPU to other applications (running on the same edge nodes) while waiting for incoming data; (ii) it creates an opportunity for DPM strategies to turn off idle CPU components to save energy.

Algorithm 2 process_minibatch ( )

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

The algorithm 2 shows the processing of a mini-batch, with the classification (lines 3-7) and training (lines 8-17). The line 4 distributes the mini-batch to each trainer. Line 5 obtains the votes for each trainer. Votes are the predictions of each model and are aggregated in line 7. The loop in line 3 can be sequential or parallel according to the characteristics of the application (e.g., classifiers with small number of operations may disable the parallelism). Then, each trainer will iterate over the mini-batch instances to calculate the weight, create the weighted instance, and train the classifier with this weighted instance (lines 8-13). ARF, SRP, and LBag, exclusively, will execute lines 14-16 as a local change detector for each classifier in the ensemble. In OBAdwin, lines 14-16 would be outside the parallel section, as the change detection is a global operation. Finally line 18 empties the mini-batch to begin accumulating again. In summary, mini-batching groups instances and reorders operations to reduce the reuse distance and the cache misses [10]). As result, mini-batching reduces both the execution time and energy consumption to process each data instance.

Despite its benefits, mini-batching has two main drawbacks as pointed out in [11]: (i) by postponing the training (in line 12 of Algorithm 2) to be executed after the classification loop (lines 3 to 6), mini-batching introduces a delay to the algorithm detect changes in data distribution; and (ii) it prevents the reuse of data structures in cache between the classification and test phases (the loops in lines 3-6, and 9-13). To circumvent these drawbacks, next we propose an improvement to mini-batching, that fuses the classification and training phases.

B. A new mini-batching strategy that fuses the classification and training phases

We propose an improvement to mini-batching as described in Algorithm 3. there are separate loops for the classification (lines 3-6) and training (lines 8-17) on the batch instances. This modification enhances cache reuse by performing both the classification and training on a single pass. In this case, the data structures (e.g., VFDT) will be loaded in the cache and reused for both the classification and training operations. Second, it reduces loop overhead. Last, in algorithm 3, the training operation is executed after all the instances of the mini-batch are classified, thus causing a short delay in model training. By fusing these phases With loop fusion instead, the classification and training occur within the same iteration (lines 3 and 6) reproducing the original behavior of the original ensemble algorithms, which perform both the classification and training operations within the same processing iteration. In the next section, we evaluate the performance of the new strategy, comparing it with the original mini-batching.

Algorithm 3 process_minibatch(...) // The new strategy that fuses the classification and training loops (MB-LF).

1: **Input**: mini-batch B
2: **for** each trainer T_i in trainers T **do in parallel**
3: **for** each instance I in B **do** ▷ Classification and training into a single loop
4: $votes$[i, j] $\leftarrow$ T_i.classify(B[j])
5: $k \leftarrow poisson(\lambda)$
6: $W_{inst} \leftarrow I * k$
7: $train_on_instance(W_{inst})$
8: **end for**
9: **if** change detected **then**
10: $reset_classifier$
11: **end if**
12: **end for**
13: $E.compile(votes)$
14: $B.clear()$

C. Hardware approaches for energy saving

We also evaluate mini-batching strategies (both the original and the improved one), by comparing them to other hardware strategies supported in commodity processors. Dynamic power management (DPM) can be programmatically enforced to save energy [15]. DPM encompasses many techniques to save energy and dynamic power. Most microprocessors nowadays can automatically turn off the clock of inactive components to this end. If there are no floating-point instructions for execution, the clock of the floating-point unit can be disabled [13]. Other techniques include putting DRAM memory and storage components in low power modes when the devices are idle. For example, DRAMs have a series of increasingly lower power modes to extend battery life in PMDs and laptops, and there have been proposals for disks that have a mode that spins more slowly when unused, to save power. Although DPM strategies act in an autonomous way, optimization strategies can be employed to reduce the number of active cores or explicitly reduce the clock frequency and save energy [15].

Core limiting (CL) is a strategy that limits the execution of the application to a subset of the available processing cores. For instance, an application may be pinned to just two out of four cores in a processor, while the remaining two cores are kept idle (thus, automatically triggering DPM mechanisms) to reduce power consumption.

A second hardware strategy supported in modern microprocessors is *voltage-frequency scaling* (VFS), which allows the user to choose among a few clock frequencies and voltages in which the microprocessor can operate. For instance, clock frequency can be adjusted by using cpufreq-utils. Power consumption is expected to grow in quadratic proportion to the increase in clock frequency, while the application performance is expected to increase linearly. The objective is to evaluate if this technique can improve energy efficiency, mainly under low-intensity

V. Experimental evaluation

This section evaluates the performance of the new strategy, compared with the original mini-batching and two hardware-based techniques. Our testbed is composed of four dedicated machines: (i) a *workload generator* reads the dataset and delivers its instances as a data stream at controlled rates; (ii) a *data stream processor* implemented by a Raspberry Pi 3 Model B with a processor of 4 cores Cortex-A53 64-bit at 1.2 GHz and 1 GB LPDDR2 memory (which is a good representative of an EC device with limited memory and processing capabilities [3]); (iii) a high precision *power sensor* collects information in real-time from the Power Distribution Unit (PDU); and (iv) a *data logger* collects all experimental data for analysis.

A. Datasets and load generation

The *data stream generator* reads the benchmark datasets and transmits a data stream over the network at controlled transmission rates of 10%, 50%, and 90% of the maximum throughput of the *data stream processor*. The objective is to evaluate the performance of the system under low-, moderate-, and high-demand workloads. We used four open access datasets[2] in the experiments:

- The *airlines* dataset is a dataset whose objective is to predict whether a given flight will be delayed, given information on the scheduled departure. The dataset was originally designed for regression, but it has also been adapted to classification by modelling the target variable as two classes: delayed, and not delayed. The dataset has 540 K instances with seven attributes, four of which are nominal.
- The *electricity* dataset was collected from the Australian New South Wales Electricity Market. The prices are not fixed, and they can be affected by the demand and supply of the market every five minutes. The goal is to identify the price changes (there are two classes: up or down) relative

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

to a moving average of the last 24h. This dataset has temporal dependencies. The dataset has 45 K normalized instances with eight attributes, one of which is nominal.

- The *give me some credit* (GMSC) dataset has data about credit scoring, and the objective is to decide whether a loan should be allowed or not. The dataset contains 150 K instances with ten attributes historical data on borrowers.
- The forest *covertype* dataset has information of forest cover type for 30 x 30 m cells obtained from the US Forest Service Region 2 resource information system (RIS). The classes correspond to different cover types. The dataset has 581K instances with nine nominal attributes, and 45 numeric attributes.The numeric attributes are binary, and there are seven imbalanced class labels.

TABLE I
SUMMARY CHARACTERISTICS DATASET.

Datasets	Airlines	GMSC	Electricity	Covertype
Instances	540K	150K	45K	581K
Features	7	10	8	54
Nominal features	4	0	1	45
Normalized	No	No	Yes	Yes

B. The algorithms used for the experiments

The *data stream processor* implements the online ensemble algorithms OzaBag, OBagASHT, OBADWIN, LBag, ARF, and SRP. We used the Massive Online Analysis (MOA) framework [3] as a baseline because it provides validated implementations of the six ensemble methods used in our experiments, and it is straightforward to extend or modify. By using MOA, our experiments can be easily reproduced and compared to other works in the literature [32]. We compared the original implementations of these algorithms in MOA, and their optimized versions. Each ensemble implements 25 HTs, which is a good trade-off between throughput, power consumption, and accuracy [11].

C. The strategies evaluated

The objective is to evaluate the performance and energy consumption of software and hardware strategies for the classification of data streams. The first experiment evaluates voltage-frequency scaling (VFS), a technique that offers a few clock frequencies and voltages in which the device can operate to use lower power. In this experiment, we executed parallel implementations (similar to the Algorithm 1) implemented in MOA (without mini-batching), just limiting the clock frequency to 600 MHz (half of the maximum frequency) running on 2 cores, and 4 cores [4]. The results are identified by the labels **FH2**, and **FH4** in Figure 1, respectively.

The second strategy evaluated is *core limiting*, a parallel implementation of the algorithms implemented in MOA executing with full clock frequency (of 1200 MHz) on 2 cores, and 4 cores. The purpose is to investigate the effect of limiting only the number of cores for power saving. For all the experiments we pinned the threads to cores for better use of caches. In particular, in the test with 2 cores, thread pinning guarantees that the idle cores can use DPM mechanisms to save power. Results for these experiments use the labels **CL2**, and **CL4**, respectively.

Finally, the third strategy evaluated is mini-batching, operating at full clock frequency (of 1200 MHz), and 4 cores. The objective is to evaluate the power savings due to software optimization. Results for this experiment are identified as **MB** in Figure 1. All the code used for the experiments is available at [1]

TABLE II
SUMMARY OF EXPERIMENTS AND THEIR PARAMETERS.

Strategy	Software optimization	Clock frequency	# of cores
S (sequential)	none	1200 (full)	1
FH2	none	600 (half)	2
FH4	none	600 (half)	4
CL2	none	1200 (full)	2
CL4	none	1200 (full)	4
MB	mini-batching	1200 (full)	4
MB-LF	MB and loop fusion	1200 (full)	4

D. Evaluating throughput and energy efficiency

The first experiment evaluates the energy efficiency and throughput of the four strategies. To accomplish this, the load generator sent instances as fast as the processor can handle, while measuring the total makespan and energy consumption for the 6 algorithms to process the entire 4 datasets. The results in Figure 1 show energy consumption is expressed as the average Joules per data instance (JPI), while the throughput is expressed as instances processed per second (IPS). Under the maximum load of the processor, the two metrics present inverse behaviors. The software strategies (MB, and MB-LF) yield throughputs higher than hardware strategies. As expected, the novel strategy (MB-LF) outperformed the pure mini-batching (MB) in throughput and energy efficiency. Mini-batching, especially with loop fusion, spends fewer processor cycles per instance, which results in lower energy consumption per instance. The strategies with half-clock frequency (FH2 and FH4) underperformed the sequential strategy (S) because the parallel implementation without mini-batching increases the cache misses.

E. Evaluating energy efficiency and latency

Next, the workload generator produced online data streams with transmission rates of 10%, 50%, and 90% of the full capacity of the stream processor. The objective is to measure energy consumption and processing latency under different levels of load intensity. Each experiment was executed for at least 5 minutes to obtain stable reads. The bars (in Fig. 2) show the energy consumption for each algorithm, and the lines show the average processing delay per instance. As a general remark, mini-batching yields the lowest energy consumption

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

[4] A parallel implementation with full clock frequency (running with four cores) is implemented by the CL4 strategy

[1] https://github.com/HPCSys-Lab/comparison-xue3m-minibatching

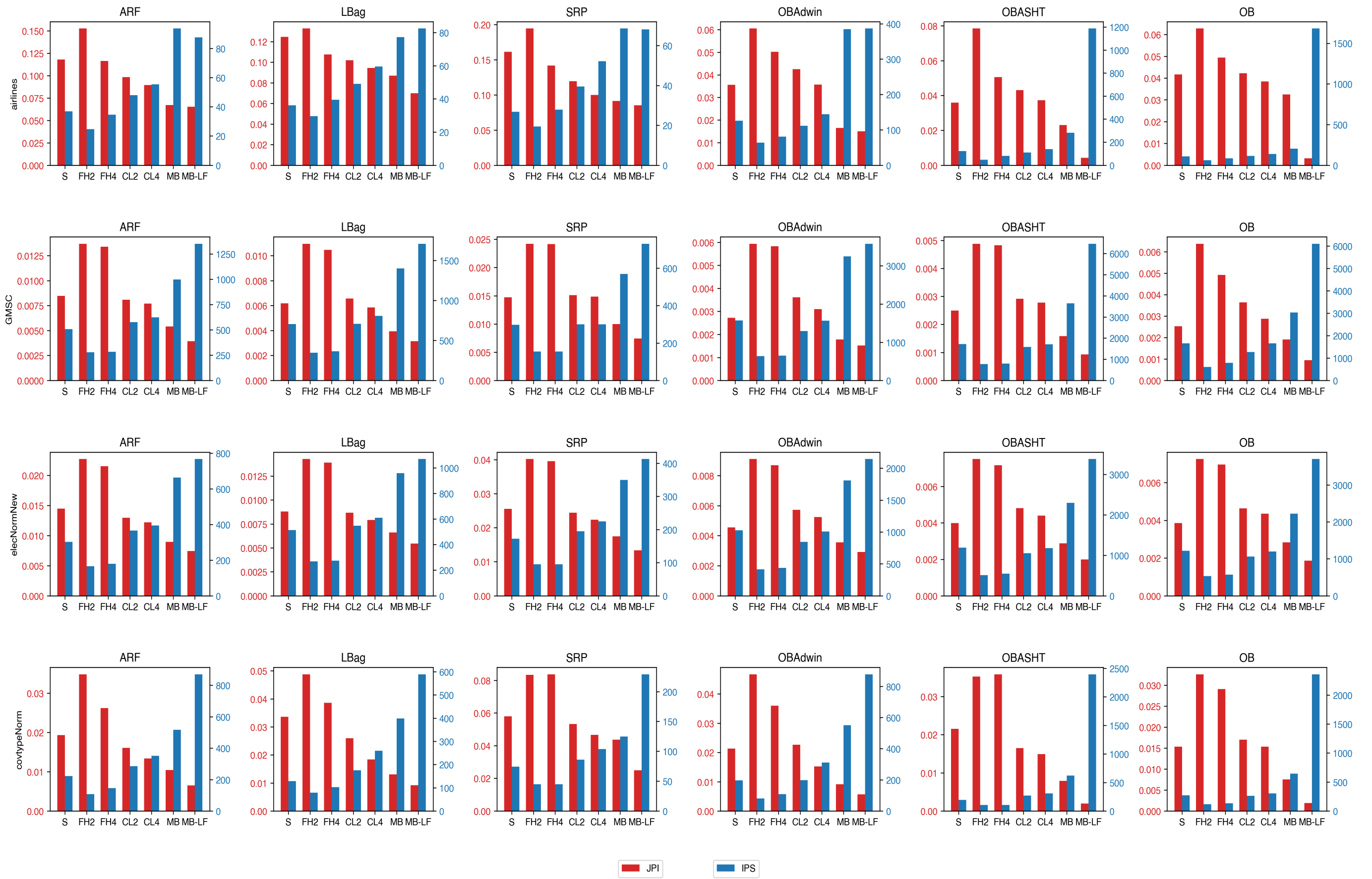

Fig. 1. Results in Joules per instance (JPI) and instances per second (IPS) for the baseline implementation in MOA (S), frequency scaling at 600 MHz and 2 cores (FH2), 600 MHz and 4 cores (FH4), 1200 MHz with 2 cores (CL2), 1200 MHz with 4 cores (CL4), mini-batching (MB), and mini-batching with loop fusion (MB-LF).

and the lowest processing delays in all scenarios evaluated. As expected, the enhanced mini-batching (MB-LF) remarkably outperforms pure mini-batching (MB).

The clock frequency downscaling (FH2, and FH4) presented the worst performance among all techniques. Although this strategy reduces the power consumption, it increases the execution times, spending more energy (Joules per instance). An additional drawback of this strategy is to decreasing the performance of other applications under execution on the same CPU. The remarkable performance of mini-batching, and its improved version with loop fusion can be explained by the benefits yielded by these strategies as discussed in section IV. In summary, these strategies (i) reduce the reuse distance (and thus, cache misses) (ii) introduce idleness periods that trigger DPM mechanisms to save energy, (iii) reduce loop overhead (by loop fusion), and (iv) avoids deferring the training to the end of the mini-batching (a drawback of pure mini-batching). As a result, software strategies that implement mini-batching can significantly reduce energy consumption in 95% of the experimental scenarios evaluated. By taking the number of data instances processed by the energy consumed (in Joules) as an energy efficiency metric, pure mini-batching improved energy efficiency by 96% on average and 169% in the best case. Likewise, mini-batching with loop fusion improved energy efficiency by 136% on average and 456% in the best case. Furthermore, software strategies support better control of the trade-offs between performance, energy efficiency, and accuracy under different load intensities.

F. Impact on predictive performance

As illustrated in Figure 3, optimisations can impact the predictive performance of the ensembles by generating slightly different trees compared to the baseline, leading to increased ensemble diversity. This is not a problem, as previous works reported that increasing diversity could lead to better ensemble stability [30]. The optimizations tested can cause disturbances in the predictive performance as operations need reordering and approximations instead of exact results. Even if the weak learners of a bagging ensemble are independent, the predictive performance can be affected because the streaming setting simulates the sampling method by randomly assigning weights to the instances based on a Poisson distribution. In the sequential algorithm, a global random number generator is responsible for assigning the weight of each instance to each learner in the ensemble. Although the weight generation is not time-consuming per se, it can add up based on the number of

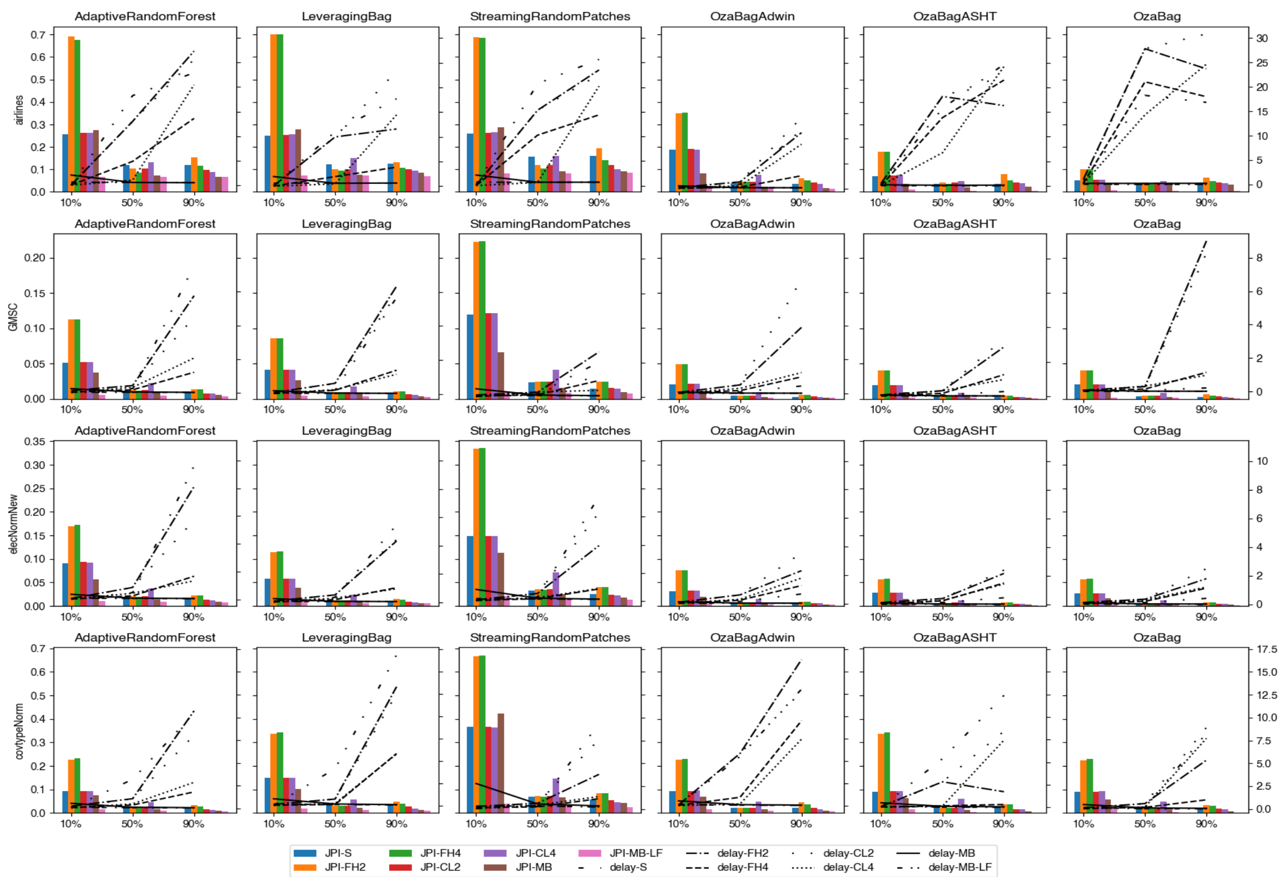

Fig. 2. Energy consumption and processing latency.

instances processed and the ensemble size. Therefore, in the improved version, this operation moves inside the thread that processes the training. In addition to increasing the parallel portion of the algorithm by using one local random number generator for each ensemble learner, it creates different instance weights to train the models. Eventually, this culminates into different cut points for the splits in the Hoeffding Trees, which increases the ensemble diversity.

VI. Conclusion

Ensemble learning has demonstrated remarkable predictive performance for the classification of data streams. Ensembles combine several single models to compose aggregated results for the classification of data streams with high predictive performance. Previous literature that analyzed the energy efficiency did not consider bagging ensembles [3], did not evaluate time measures [8], or did not evaluate hardware-based strategies [11]. We also addressed previous limitations of the mini-batching proposed in [11]. 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 process four datasets under different load intensities in a distributed testbed. Our experiments show that mini-batching outperforms the hardware strategies in terms of performance and energy efficiency in 95.7% of the tested cases, indicating that it works for all bagging algorithms. More precisely, the pure mini-batching (proposed in [11] improved energy efficiency (i.e., the amount of work performed by the same energy consumed) by 96% on average and 169% in the best case, while our new mini-batching strategy which fuses the classification and training phases improved energy efficiency by 136% on average and 456% in the best case. Furthermore, as a general remark, we show that software strategies support better control of the trade-offs between performance, energy efficiency, and accuracy under different load intensities. As future work, we plan to investigate adaptive mini-batching settings to balance latency, energy consumption and accuracy under varying loads.

References

[1] W. Z. Khan, E. Ahmed, S. Hakak, I. Yaqoob, and A. Ahmed, “Edge computing: A survey,” *Future Generation Computer Systems*, vol. 97, pp. 219–235, 2019.

[2] X. Yu, C. Cecati, T. Dillon, and M. G. Simo, “An industrial electronics perspective,” *IEEE Industrial Electronics Magazine*, vol. 5, no. 9, pp. 49–63, 2011.

[3] J. Lopes, E. J. Santana, V. G. da Costa, B. Zarpelão, and S. Barbon Junior, “Evaluating the four-way performance trade-off for data stream

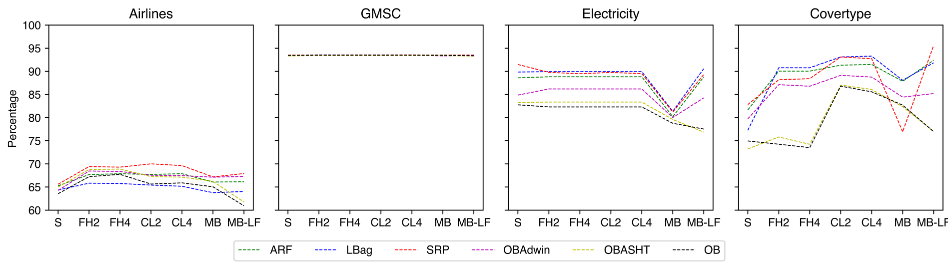

Fig. 3. Accuracy for all the strategies with MB-LF.

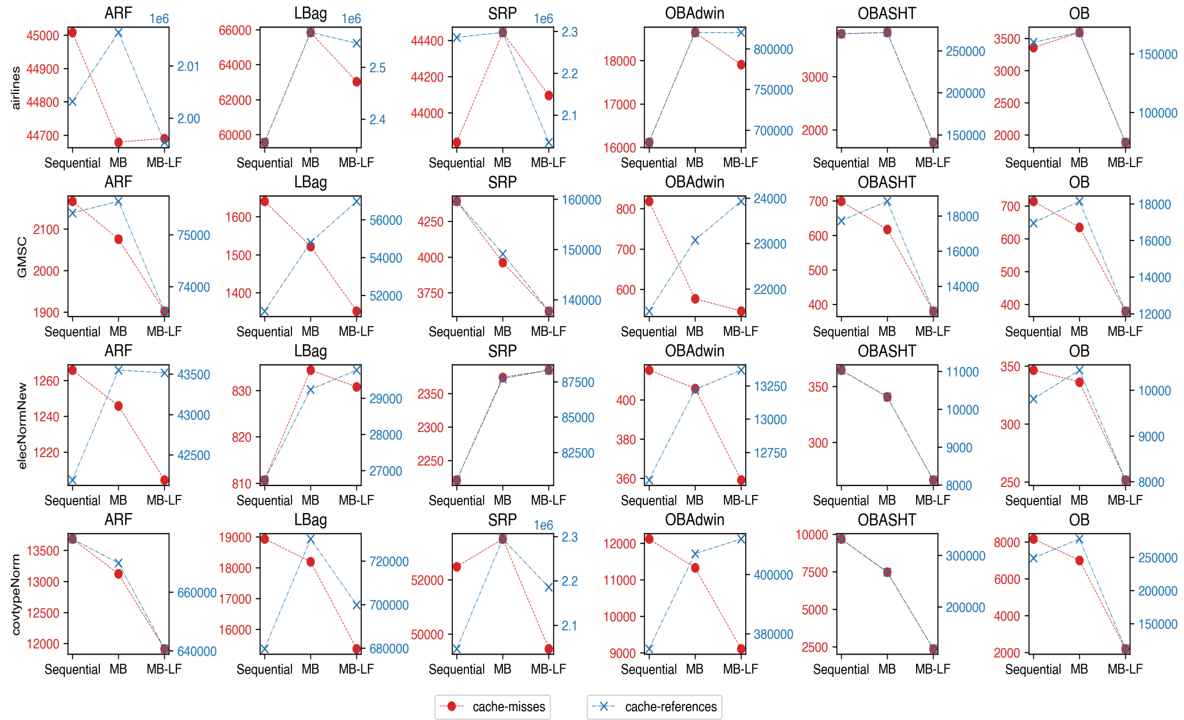

Fig. 4. The cache-references accounts for data requests missed in the L1 and L2 caches. The best scenario is when the two lines decrease (from left to right) between MB and MB-LF. The worst scenario is when two lines increase (as in SRP/Elec).

classification in edge computing," *IEEE Transactions on Network and Service Management*, vol. 17, no. 2, pp. 1013–1025, 2020.

[4] A. Shakarami, M. Ghobaei-Arani, and A. Shahidinejad, "A survey on the computation offloading approaches in mobile edge computing: A machine learning-based perspective," *Computer Networks*, vol. 182, p. 107496, dec 2020.

[5] X. Jin, L. Li, F. Dang, X. Chen, and Y. Liu, "A survey on edge computing for wearable technology," *Digital Signal Processing: A Review Journal*, vol. 1, p. 103146, 2021.

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

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

[8] E. García-Martín, A. Bifet, and N. Lavesson, "Energy modeling of hoeffding tree ensembles," *Intelligent Data Analysis*, vol. 25, no. 1, pp. 81–104, 2021.

[9] 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 *IEEE Intl Conf. on High Performance Computing and Communications (HPCC)*, 2020.

[10] G. Cassales, H. Gomes, A. Bifet, B. Pfahringer, and H. Senger, "Improving the performance of bagging ensembles for data streams through mini-batching," *Information Sciences*, vol. 580, pp. 260–282, 2021.

[11] G. Cassales, H. Gomes, A. Bifet, B. Pfahringer, and H. Senger, "Balancing Performance and Energy Consumption of Bagging Ensembles for the Classification of Data Streams in Edge Computing," *IEEE Transactions on Network and Service Management*, Dec. 2022.

[12] A.-C. Orgerie, M. D. d. Assuncao, and L. Lefevre, "A survey on techniques for improving the energy efficiency of large-scale distributed systems," *ACM Computing Surveys*, vol. 46, no. 4, pp. 1–31, 2014.

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

[14] D. C. Snowdon, S. Ruocco, and G. Heiser, "Power management and dynamic voltage scaling: Myths and facts," in *Workshop on Power Aware Real-time Computing, New Jersey, USA*, vol. 31, p. 34, Citeseer, 2005.

[15] D. A. Coutinho, D. de Sensi, A. F. Lorenzon, K. Georgiou, J. Nunez-

Yanez, K. Eder, and S. Xavier-De-Souza, "Performance and energy trade-offs for parallel applications on heterogeneous multi-processing systems," *Energies*, vol. 13, may 2020.

[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 *2015 IEEE/ACM Intl. Conf. Advances in Social Networks Analysis and Mining (ASONAM)*, pp. 1125–1132, 2015.

[18] V. G. T. da Costa, A. C. P. d. L. F. de Carvalho, and S. Barbon Junior, "Strict Very Fast Decision Tree: A memory conservative algorithm for data stream mining," *Pattern Recognition Letters*, vol. 116, pp. 22–28, 2018.

[19] V. G. T. da Costa, E. J. Santana, J. F. Lopes, and S. Barbon, "Evaluating the four-way performance trade-off for stream classification," in *Intl.ernational Conference on Green, Pervasive, and Cloud Computing*, pp. 3–17, Springer, 2019.

[20] P. Domingos and G. Hulten, "Mining high-speed data streams," in *Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 71–80, ACM SIGKDD, Sep. 2000.

[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 Computing Surveys*, vol. 46, no. 1, pp. 1–31, 2013.

[22] L. Breiman, "Bagging predictors," *Machine Learning*, vol. 24, no. 2, pp. 123–140, 1996.

[23] N. C. Oza and S. J. Russell, "Online bagging and boosting," in *International Workshop on Artificial Intelligence and Statistics*, pp. 229–236, PMLR, 2001.

[24] C. Manapragada, G. I. Webb, and M. Salehi, "Extremely fast decision tree," in *ACM Intl. Conference on Knowledge Discovery & Data Mining*, pp. 1953–1962, 2018.

[25] A. Bifet, G. Holmes, B. Pfahringer, R. Kirkby, and R. Gavaldà, "New ensemble methods for evolving data streams," in *Proc. 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 139–148, 2009.

[26] A. Bifet and R. Gavaldà, "Learning from time-changing data with adaptive windowing," in *SIAM Intl. Conf. on Data Mining*, vol. 7, 2007.

[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.), pp. 135–150, Springer Berlin Heidelberg, 2010.

[28] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[29] H. M. Gomes, A. Bifet, J. Read, J. P. Barddal, F. Enembreck, B. Pfharinger, G. Holmes, and T. Abdessalem, "Adaptive random forests for evolving data stream classification," *Machine Learning*, vol. 106, no. 9, pp. 1469–1495, 2017.

[30] H. M. Gomes, J. Read, and A. Bifet, "Streaming random patches for evolving data stream classification," in *IEEE Intl. Conf. on Data Mining*, pp. 240–249, 2019.

[31] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters," in *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*, pp. 10–10, USENIX Association, 2012.

[32] A. Bifet, G. Holmes, B. Pfahringer, P. Kranen, H. Kremer, T. Jansen, and T. Seidl, "MOA: Massive online analysis, a framework for stream classification and clustering," in *Workshop on Applications of Pattern Analysis*, pp. 44–50, PMLR, 2010.