Improving Performance and Energy Efficiency of the Classification of Data Streams on Edge Computing

Reginaldo Luna[1] **, Guilherme Cassales** [2]**, Hermes Senger** [1]

[1]Departamento de Computação - Universidade Federal de São Carlos

[2]University of Waikato, New Zealand

76regi@gmail.com, hermes@ufscar.br, guilherme.cassales@waikato.ac.nz

Abstract. *In this work, we propose a loop transformation to improve performance and energy efficiency of ensembles. We compare the performance of our technique with three other strategies for improving energy efficiency and throughput in data stream classification using six state-of-the-art ensemble algorithms and four benchmark datasets. Our results show that software strategies can significantly reduce energy consumption. 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.*

1. Introduction

Edge Computing (EC) is a paradigm that moves the services and utilities of cloud computing closer to data sources (e.g., sensors) [Khan et al. 2019]. Ensembles of classifiers are examples of ML techniques that demonstrated noticeable accuracy for the classification of data streams [Gomes et al. 2017] by using several weak learners which can produce more accurate results. *Mini-batching* is a strategy that groups data instances in a data stream, to be processed together. This technique demonstrated to improve performance, and energy efficiency of parallel implementations of bagging ensembles for the classification of data streams [Cassales et al. 2022]. In this work, we propose an improvement to the original *mini-batching* strategy, which applies loop transformations to improve the performance and energy efficiency of ensembles of classifiers. We compare the performance of our technique with three other strategies: pure *mini-batching*, Dynamic Power Management (DPM), and Voltage-Frequency Scaling (VFS).

2. 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 $\rightarrow \infty$) [Silva et al. 2013]. 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 (HT) [Domingos and Hulten 2000], an incremental tree designed to cope with massive data streams. Despite being able to make splits with limited data, its limitations are exposed when attempting to model complex learning problems with a single tree. To overcome this issue, a popular strategy is to ensemble several models using Bagging [Breiman 1996]. Bagging is a ML technique that involves creating multiple models by training them on different subsets of the original training data, and then combining their predictions to make a final prediction.

3. Improvement mini-batching with loop fusion for energy saving

The algorithm 1 shows the processing of a mini-batch. It performs the classification (lines 2-6) and training (lines 7-16). The line 3 distributes the mini-batch to each trainer. Line 4 obtains the votes for each trainer. Votes are the predictions of each model and are aggregated in line 6. The loop in line 2 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 all the mini-batch instances to calculate the weight, create the weighted instance, and train the classifier with this weighted 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.

The algorithm 2 merges the loops for the classification and training in one loop (*loop fusion*). This improves cache reuse by performing both the classification and training on a single pass. In this case, the data structures will be loaded in cache and reused for classification and training operations.

Algorithm 1 process_minibatch (...) // As proposed in [Cassales et al. 2021]

```
 1: Input: mini-batch $B$
 2: for each trainer $T_i$ in trainers $T$ do in parallel        ▷ The classification loop
 3:     $T_i.instances \leftarrow B$
 4:     $votes_i \leftarrow T_i.classify(T_i.instances)$
 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                ▷ The training loop
 9:         $k \leftarrow poisson(\lambda)$
10:         $W\_inst \leftarrow I * k$
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: $B.clear()$
```

4. Experimental evaluation

Our testbed is composed of four dedicated machines in an isolated network: (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; (iii) a high precision *power sensor* Yokogawa MW-100 collects information in real-time from the Power; and (iv) a *data logger* collects all experimental data for analysis.

5. Evaluating throughput and energy efficiency

We compare our new strategy with three other strategies: (i) the *Core Limiting* (CL) limits (i.e., pin) the execution of the application to a subset of the available processing cores. (ii)

the *Voltage-Frequency Scaling* (VFS) strategy, which allows to reduce clock frequencies and voltages to save energy; (*iii*) the pure *Mini-Batching* (MB). A load generator sent instances at the maximum rate the processor can handle while measuring the total makespan and energy consumption for the 6 algorithms OzaBag, OBagASHT, OBADWIN, LBag, ARF, and SRP to process the entire 4 datasets. The results in Figure 1 show the energy consumption in average Joules Per Data Instance (JPI), expressed by the mathematical formula $JPI = \frac{E}{n}$, where n is the number of processed instances and E is the amount of energy expended during processing, while the throughput is expressed as Instances Processed Per Second (IPS) with the following equation $IPS = \frac{n}{s}$, where n is the number of processed instances and s is the processing time. The software strategies (MB and MB-LF) presented higher throughput than the 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.

Algorithm 2 process_minibatch(...) // New version with loop fusion

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

6. Conclusion

Our experiments show that mini-batching outperforms the hardware strategies in terms of throughput and energy efficiency in all of the tested cases. Pure mini-batching 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. Likewise, mini-batching with loop fusion (proposed in this paper) improved energy efficiency by 136% on average and 456% in the best case. Furthermore, software strategies support better control of the application of the optimizations without affecting other applications that execute on the same hardware. As future work we evaluate impact in predictive performance and efficiency latency and plan to investigate adaptive mini-batching settings to balance latency, energy consumption and accuracy under varying loads.

7. Acknowledgements

We would like acknowledgements to the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) for their support to research in this project.

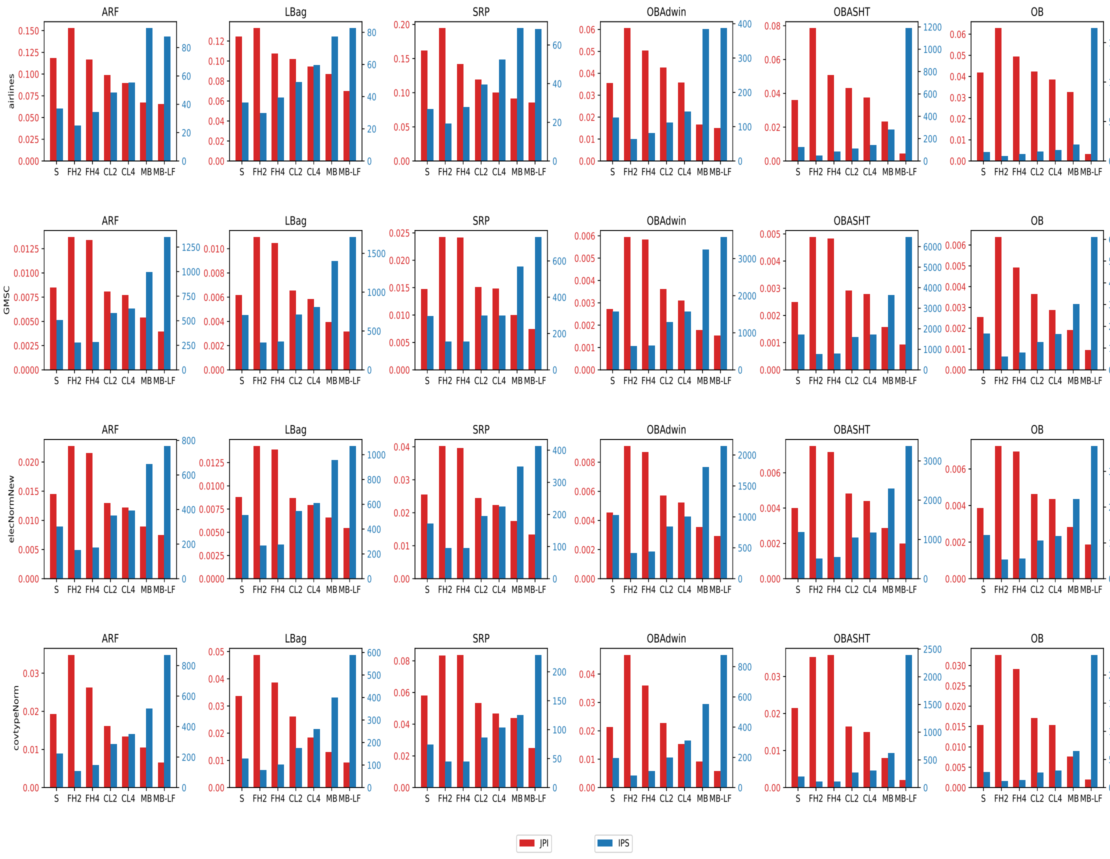
Figure 1. Results in Joules per instance (JPI), and instances per second (IPS).

References

Breiman, L. (1996). Bagging predictors. *Machine Learning*, 24(2):123–140.

Cassales, G., Gomes, H., Bifet, A., Pfahringer, B., and Senger, H. (2021). Improving the performance of bagging ensembles for data streams through mini-batching. *Information Sciences*, 580:260–282.

Cassales, G., Gomes, H., Bifet, A., Pfahringer, B., and Senger, H. (2022). Balancing Performance and Energy Consumption of Bagging Ensembles for the Classification of Data Streams in Edge Computing. *IEEE Trans. on Network and Service Management*.

Domingos, P. and Hulten, G. (2000). Mining high-speed data streams. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 71–80. ACM SIGKDD.

Gomes, H. M., Barddal, J. P., Enembreck, F., and Bifet, A. (2017). A survey on ensemble learning for data stream classification. *ACM Computing Surveys*, 50(2):1–36.

Khan, W. Z., Ahmed, E., Hakak, S., Yaqoob, I., and Ahmed, A. (2019). Edge computing: A survey. *Future Generation Computer Systems*, 97:219–235.

Silva, J. A., Faria, E. R., Barros, R. C., Hruschka, E. R., Carvalho, A. C. d., and Gama, J. (2013). Data stream clustering: A survey. *ACM Computing Surveys*, 46(1):1–31.