

# Energy modeling of Hoeffding tree ensembles

Eva García-Martín<sup>a,\*</sup>, Albert Bifet<sup>b</sup> and Niklas Lavesson<sup>c</sup>

<sup>a</sup>*Department of Computer Science, Blekinge Institute of Technology, Karlskrona, Sweden*

<sup>b</sup>*Télécom ParisTech, Paris, France*

<sup>c</sup>*Jönköping University, Jönköping, Sweden*

**Abstract.** Energy consumption reduction has been an increasing trend in machine learning over the past few years due to its socio-ecological importance. In new challenging areas such as edge computing, energy consumption and predictive accuracy are key variables during algorithm design and implementation. State-of-the-art ensemble stream mining algorithms are able to create highly accurate predictions at a substantial energy cost. This paper introduces the *nmin adaptation* method to ensembles of Hoeffding tree algorithms, to further reduce their energy consumption without sacrificing accuracy. We also present extensive theoretical energy models of such algorithms, detailing their energy patterns and how *nmin adaptation* affects their energy consumption. We have evaluated the energy efficiency and accuracy of the *nmin adaptation* method on five different ensembles of Hoeffding trees under 11 publicly available datasets. The results show that we are able to reduce the energy consumption significantly, by 21% on average, affecting accuracy by less than one percent on average.

Keywords: GreenAI, Hoeffding trees, data stream mining, energy efficiency, ensembles

## 1. Introduction

Energy consumption in machine learning is starting to gain importance in state-of-the-art research. This is clearly visible in areas where researchers are incorporating the inference or training of the model inside the device (i.e. edge computing or edge AI). An area whose main focus is on real-time prediction on the edge is data stream mining.

One of the most well-known and used classifier is the decision tree, due to its explainability advantage. In the data stream setting, where we can only do one pass over the data, and we can not store all of it, the main problem of building a decision tree is the need to reuse the examples to compute the best splitting attributes. Hulten and Domingos [16] proposed the Hoeffding Tree or VFDT, a very fast decision tree for streaming data, where instead of reusing instances, we wait for new instances to arrive. The most interesting feature of the Hoeffding tree is that it builds an identical tree with a traditional one, with high probability if the number of instances is large enough, and that it has theoretical guarantees about that.

Decision trees are usually not used alone, but within ensembles methods. Ensemble methods are combinations of several models whose individual predictions are combined in some manner (e.g., averaging or voting) to form a final prediction. They have several advantages over single classifier

---

\*Corresponding author: Eva García-Martín, Department of Computer Science, Blekinge Institute of Technology, Karlskrona, Sweden. E-mail: eva.garcia.martin@bth.se.

methods: they are easy to scale and parallelize, they can adapt to change quickly by pruning under-performing parts of the ensemble, and they therefore usually also generate more accurate concept descriptions.

Advancements in data stream mining have been primarily focused on creating algorithms that output higher predictive performance. For that, they used ensembles of existing algorithms. However, these solutions output high predictive performance at the cost of higher energy consumption. We recently conducted an experiment comparing Online Boosting [34,35] to standard Hoeffding trees for the Forest dataset (Section 5.1). Online Boosting was able to achieve 14% more accuracy but consuming 5 times more energy. To address this challenge, we present the *nmin adaptation* method for ensembles of Hoeffding Tree algorithms [16], to make them more feasible to run in the edge.

The *nmin adaptation* method is a method presented in [18], which reduces the energy consumption of standard Hoeffding tree algorithms by adapting the number of instances needed to create a split. We extend that study by incorporating the *nmin adaptation* method to ensembles of Hoeffding trees. The goal of this paper is two-fold:

- Present an energy efficient approach to real-time prediction with high levels of accuracy.
- Present detailed theoretical energy models for ensemble of Hoeffding trees, together with a generic approach to create energy models, applicable to any class of algorithms.

We have conducted experiments on five different ensembles of Hoeffding trees (Leveraging Bagging [6], Online Coordinate Boosting [37], Online Accuracy Updated Ensemble [11], Online Bagging [36], and Online Boosting [36]), with and without *nmin adaptation*, on 11 publicly available datasets. The results show that we are able to reduce the energy consumption by 21%, affecting accuracy by less than 1%, on average.

This approach achieves similar levels of accuracy as state-of-the-art ensemble online algorithms, while significantly reducing its energy consumption. We believe this is a significant step towards a greener data stream mining, by proposing not only more energy efficient algorithms, but also creating a better understanding of how ensemble online algorithms consume energy.

The rest of the paper is organized as follows: The work related to this study is presented in Section 2. Background on Hoeffding trees and the *nmin adaptation* method is presented in Section 3. The main contribution of this paper is presented in Section 4. There we first describe how to build energy models for any kind of algorithm (Fig. 2), we then present extensive theoretical energy models for the Online Bagging, Leveraging Bagging, Online Boosting, Online Coordinate Boosting, and Online Accuracy Updated Ensemble algorithms. This helps at understanding the energy bottlenecks of the algorithm, and why *nmin adaptation* is efficient at handling those bottlenecks. The design of the experiments together with the results are presented in Sections 5 and 6 respectively. The paper ends with conclusions in Section 7.

## 2. Related work

Energy efficiency has been widely studied in the field of computer architecture for decades [23]. Moore's law stated that the performance of CPUs was going to double every 18 months, by doubling the number of transistors. This prediction was made considering one fundamental constraint, that the energy consumed by each unit of computing would decrease as the number of transistors increased.<sup>1</sup> The

<sup>1</sup><https://www.forbes.com/2010/04/29/moores-law-computing-processing-opinions-contributors-bill-dally.html>.

CPU scaling from Moore does not apply anymore, and the main reason is because there was not enough focus on scaling the power while increasing performance [25]. This has created a paradigm shift, where power is the key factor studied to improve computer performance, creating processors that handle more operations using the same amount of power [28].

Machine learning research, and in particular deep learning, is aware of the need to focus not only on creating more accurate models, but also on creating energy efficient models [20,21,38]. The researchers in this area have realized that the amount of energy and resources (e.g. GPUs) needed to perform training and inference on such models is impractical, especially for mobile platforms. Currently most of the machine learning models used in the phone are accessed through the cloud, since doing inference in the phone is challenging. Some research is going in that direction, such as *The Low Power Image Recognition Challenge* (LPIRC) [19], and the work by [13,39].

Approaches to mine streams of data have been evolving during the past years. We consider the Hoeffding tree algorithm [16] to be the first algorithm that was able to classify data from an infinite stream of data in constant time per example. That approach was later improved to be able to handle concept drift [26], that is, non-stationary streams of data that evolve through time. The ability to handle concept drift introduces many computational demands, since the algorithm needs to keep track of the error to detect when a change occurs. ADWIN (adaptive windowing) [3] is the most efficient change detector that can be incorporated to any algorithm. The Hoeffding Adaptive Tree (HAT) [4] was introduced as an extension to the standard Hoeffding tree algorithm that is able to handle concept drift using the ADWIN detector.

In relation to energy efficiency and data stream mining, several studies have recently focused on creating more energy efficient versions of existing models. The Vertical Hoeffding Tree (VHT) [29] was introduced to parallelize the induction of Hoeffding trees. Another approach for distributed systems is the Streaming Parallel Decision Tree algorithm (SPDT) [2]. Marrón et al. [33] propose a hardware approach to improve Hoeffding trees, by parallelizing the execution of random forests of Hoeffding trees and creating specific hardware configurations. Another streaming algorithm that was improved in terms of energy efficiency was the KNN version with self-adjusting memory [30].

A recent publication at the International Conference on Data Mining [31] presented an approach similar to ours, which estimates the value of  $nmin$  to avoid unnecessary split attempts. Although their approach seems to better estimate the value of  $nmin$ , our approach is still more energy efficient (based on their results), which is the ultimate goal of our study. We believe that trading off a few percentages of accuracy is necessary in some scenarios (e.g. embedded devices) where energy and battery consumption is the main concern. This work is an extension of an already published study where we introduced *nmin adaptation* [18]. While in that work the *nmin adaptation* method was only applied to the standard version of the Hoeffding tree algorithm, this study proposes more energy efficient approaches to create ensembles of Hoeffding trees, validated by the experiments on five different algorithms and 11 different datasets. On top of that, this study presents, to the best of our knowledge, the first approach to create generic energy model for any class of machine learning algorithm.

### 3. Background

This section explains in detail Hoeffding trees, together with the ensembles of Hoeffding trees used in this study.

### 3.1. Hoeffding tree algorithm

The Hoeffding tree algorithm (Algorithm 2), also known as Very Fast Decision Tree (VFDT) [16], was first introduced in the year 2000, presenting the first approach that was able to mine from a stream of data in constant time per example, with low computational constraints, and being able to read the data only once without storing it. The algorithm builds and updates the model as the data arrive, storing only the necessary statistics at each node to be able to grow the tree.

The algorithm reads an instance, traverses the tree until it reaches the corresponding leaf, and updates the statistics at that leaf based on the information from that instance. For attributes with discrete values, the statistics are counts of class values for each attribute value. On the other hand, for attributes with numerical values there are several approaches to save the information in the most efficient way. The most common approach used nowadays is to maintain a Gaussian function with the mean, standard deviation, maximum, and minimum, at each node, for each class label and attribute.

Once  $nmin$  instances are read in a particular leaf, the algorithm calculates the information gain (entropy) for each attribute, using the aforementioned statistics. If  $\Delta\bar{G} > \varepsilon$ , i.e. the difference between the information gain from the best and the second best attribute ( $\Delta\bar{G}$ ) is higher than the Hoeffding Bound [24]( $\varepsilon$ ), then a split occurs, and the leaf substituted by the node with the best attribute. The Hoeffding bound( $\varepsilon$ ) is defined as:

$$\varepsilon = \sqrt{\frac{R^2 \ln(1/\delta)}{2n}} \quad (1)$$

and it states that the split on the best attribute having observed  $n$  number of examples, will be the same as if the algorithm had observed an infinite number of examples, with probability  $1 - \delta$ . The idea is that the Hoeffding tree can be approximated to a standard decision tree by observing a sufficient number of instances at each node, to create a confident split. On the other hand, if  $\Delta\bar{G} < \varepsilon < \tau$ , a tie occurs. This happens when the two top attributes are equally good, thus the tree splits on any of those.

---

**Algorithm 1** AttemptSplit **Symbols:**  $X$ : attributes;  $\varepsilon$ : Hoeffding bound Eq. (1)

---

- 1: Compute  $\bar{G}_l(X_i)$  for each attribute  $X_i$
  - 2:  $\Delta\bar{G} = \bar{G}_l(X_a) - \bar{G}_l(X_b)$  ▷ Difference between two best attributes
  - 3: **if** ( $\Delta\bar{G} > \varepsilon$ ) or ( $\varepsilon < \tau$ ) **then**
  - 4:     Split  $\leftarrow$  True
  - 5: **end if**
- 

### 3.2. $nmin$ adaptation

The  $nmin$  adaptation method was previously introduced in [18]. While that study only focused on the energy reduction of standard Hoeffding tree algorithms, this study proposes to use the  $nmin$  adaptation method on ensembles of Hoeffding trees, to align with the current approaches in data stream mining.

The goal of the  $nmin$  adaptation method is to estimate the number of instances ( $nmin$ ) that are needed to make a split with confidence  $1 - \delta$ . Current Hoeffding tree algorithms use a fixed value of  $nmin$  instances. Thus the batch of instances that are observed on each node before checking for a possible split is the same for each node, and during the complete execution of the algorithm. As is explained in Section 4.2.1, having a fixed  $nmin$  value is energy inefficient. This is because there are many times where those instances are not enough to make a confident split, thus all those computations are done unnecessarily.

Our method estimates, for each node, the  $nmin$  instances that ensure a split, to calculate the splitting attributes only when there is going to be a split. Since each node will have their own  $nmin$ , we allow the

---

**Algorithm 2** Hoeffding Tree. **Symbols:**  $HT$ : Initial tree;  $X$ : set of attributes;  $G(\cdot)$ : split evaluation function;  $\tau$ : hyperparameter used for attributes with tied information gain;  $nmin$ : hyperparameter to decide when to check for a split.

---

```

1: while stream is not empty do
2:   Read instance
3:   Traverse the tree using  $HT$ 
4:   Update statistics at leaf  $l$ 
5:   Increment  $n_l$ : instances seen at  $l$ 
6:   if  $nmin \leq n_l$  then
7:     AttemptSplit( $l$ )
8:     if Split==True then
9:       CreateChildren( $l$ )
10:    else
11:      Disable attr  $\{X_p | (\overline{G}_l(X_p) - \overline{G}_l(X_a)) > \varepsilon\}$ 
12:    end if
13:  end if
14: end while

```

$\triangleright$  Until leaf  $l$  is reached  
 $\triangleright$  Nominal and numerical attributes  
 $\triangleright$  Call Function from Algorithm 1  
 $\triangleright$  New leaf  $l_m$  with initialized statistics

---

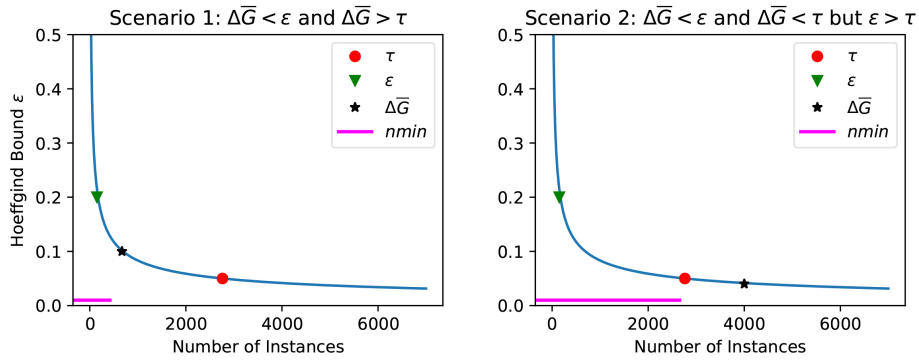


Fig. 1. Example of the  $nmin$  adaptation method. The value of  $nmin$  is going to be adapted based on two scenarios: scenario 1 (left plot), and scenario 2 (right plot).

tree to grow faster (lower  $nmin$ ) in those branches where there is a clear attribute to split on, and to delay the growth (higher  $nmin$ ) in those branches where there is no clear attribute to split on.

The value of  $nmin$  is going to be adapted only when a non-split occurs, otherwise we assume that the current value is the optimal one. The method follows two approaches, illustrated in Fig. 1.

The first approach (scenario 1) approximates the value of  $nmin$  when the non-split occurred because no attribute was the clear winner (Fig. 1 left plot). In this case,  $nmin$  is estimated as the number of instances needed for the best attribute to be higher than the second best attribute with a difference of  $\varepsilon$ . Taking a look at the left plot from Fig. 1,  $nmin$  are the instances needed for the green triangle ( $\varepsilon$ ), to reach the black star ( $\Delta\overline{G}$ ). When we wait for those  $nmin$  instances, then  $\Delta\overline{G} > \varepsilon$ , satisfying the condition for a split. More formally, for the first scenario:

$$nmin = \left\lceil \frac{R^2 \cdot \ln(1/\delta)}{2 \cdot (\Delta G)^2} \right\rceil \quad (2)$$

The second approach (scenario 2) approximates the value of  $nmin$  when the non-split occurred because  $\varepsilon > \tau$ . In this scenario, although the attributes are very similar ( $\Delta\overline{G} < \varepsilon$ ), and their difference in entropy is smaller than  $\tau$ , the confidence is still not high enough to make a split. Taking a look at the right plot from Fig. 1,  $nmin$  are the instances needed for the green triangle ( $\varepsilon$ ), to reach the red dot ( $\tau$ ). When we wait for those instances, then  $\varepsilon < \tau$ , satisfying the condition to create a split. More formally, the  $nmin$  approximation for the scenario 2 is defined as:

$$nmin = \left\lceil \frac{R^2 \cdot \ln(1/\delta)}{2 \cdot \tau^2} \right\rceil \quad (3)$$

The *nmin adaptation* implemented for the Hoeffding tree algorithm is portrayed in Algorithm 3.

---

**Algorithm 3** Hoeffding Tree with *nmin* adaptation. **Symbols:** *nmin*: hyperparameter initially set by the user that is going to be adapted; *HT*: Initial tree; *X*: set of attributes; *G*(·): split evaluation function;  $\tau$ : hyperparameter used for attributes with tied information gain

---

```

1: while stream is not empty do
2:   Read instance
3:   Traverse the tree using HT ▷ Until leaf l is reached
4:   Update statistics at leaf l ▷ Nominal and numerical attributes
5:   Increment  $n_l$ : instances seen at l
6:   if nmin ≤  $n_l$  then
7:     AttemptSplit(l) ▷ Call Function from Algorithm 1
8:     if Split==True then
9:       CreateChildren(l) ▷ New leaf  $l_m$  with initialized statistics
10:    else
11:      Disable attr  $\{X_p | (|\bar{G}_l(X_p) - \bar{G}_l(X_a)| > \varepsilon)\}$  ▷ Adapt the value of nmin
12:      if  $\Delta \bar{G} \leq \tau$  then
13:         $nmin = \left\lceil \frac{R^2 \cdot \ln(1/\delta)}{2 \cdot \tau^2} \right\rceil$  ▷ Scenario 2
14:      else
15:         $nmin = \left\lceil \frac{R^2 \cdot \ln(1/\delta)}{2 \cdot (\Delta G)^2} \right\rceil$  ▷ Scenario 1
16:      end if
17:    end if
18:  end if
19: end while

```

---

## 4. Energy modeling of Hoeffding tree ensembles

This section presents the main contribution of this paper, guidelines on how to construct energy efficient ensembles of Hoeffding trees, validated with the experiments on Section 6. In order to reduce the energy consumption of ensembles of Hoeffding trees, we apply the *nmin* adaptation method to the Hoeffding tree algorithm, and use that algorithm as the base for the different ensembles. To have a more detailed view on how energy is consumed on ensembles of Hoeffding trees, we first portray an approach to create generic theoretical energy models for any kind of algorithm (Section 4.1). We then show how that generic approach applied for ensemble of Hoeffding trees (Section 4.2 and Fig. 3). That section finalizes with a detailed energy model and explanation of each algorithm: Online Bagging, Online Boosting, Leveraging Bagging, Online Coordinate Boosting, Online Accuracy Updated Ensemble, and Hoeffding tree with *nmin adaptation*.

### 4.1. Generic approach to create energy models

We have summarized the different steps to create theoretical energy models in Fig. 2.

First, *Step 1* formalizes the generic energy model, as a sum of the different operations (mapped then to algorithm functions) categorized by type of operation. Thus, the total energy of a model can be expressed as:

$$E = n_{FPU} \cdot E_{FPU} + n_{INT} \cdot E_{INT} + n_{cache} \cdot E_{cache} + n_{cache\_miss} \cdot (E_{cache\_miss} + E_{DRAM})$$

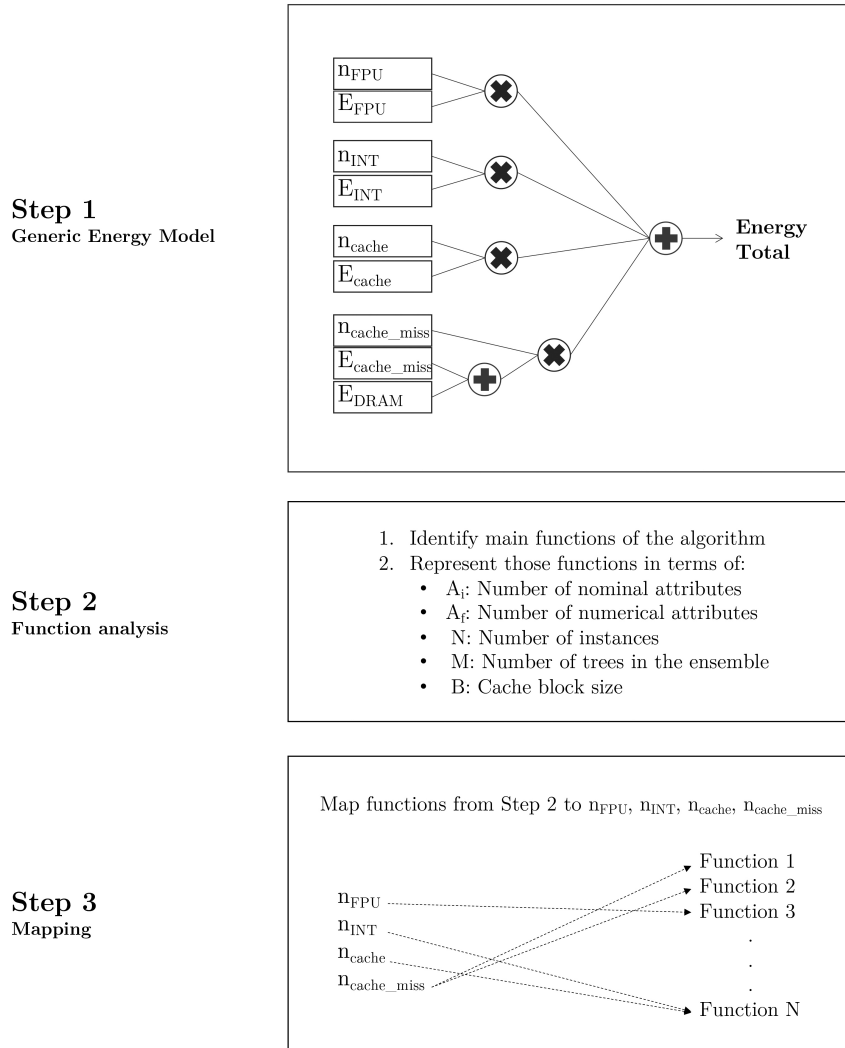


Fig. 2. Generic approach to build energy models for any kind of algorithm.

The operations are divided into integer, floating point operations, and DRAM and cache accesses. We consider a DRAM access everytime a cache miss occurs. The energy per access or per computation ( $E_{FPU}$ ,  $E_{INT}$ ,  $E_{cache\_miss}$ ,  $E_{DRAM}$ ) will depend on the specific hardware where the algorithm is run. However, knowing which type of operations consumes more energy (example a DRAM access consumes 100 times more energy than a floating point operation), together with the amount of operations for each type, gives a detailed overview of the theoretical energy consumption of the algorithm. *Step 1* can be done only once, as appears in the figure, and can be used for any algorithm. It can be adapted to specific hardware components if needed.

*Step 2* focuses on identifying the main functions of the algorithm, and then representing them in terms of generic features such as number of attributes or size of the ensemble.

In *Step 3* the goal is to map the functions from *Step 2* to the different type of operations. For example, a function that counts the number of instances can be expressed as one integer operation per instance.

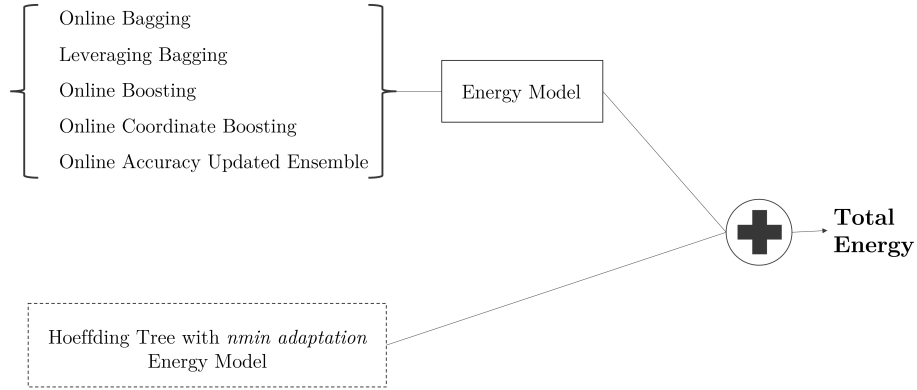


Fig. 3. Energy model approach for ensembles of Hoeffding trees with *nmin* adaptation. Steps: i) Obtain the energy model (following Fig. 2) for the general ensemble of trees; ii) Obtain the energy model of Hoeffding trees with *nmin* adaptation; iii) Sum the energy model variables. The energy model for Hoeffding trees (dashed box) can be substituted by the energy model of any other algorithm that is going to be part of the ensembles (for example Hoeffding Adaptive Trees [4]).

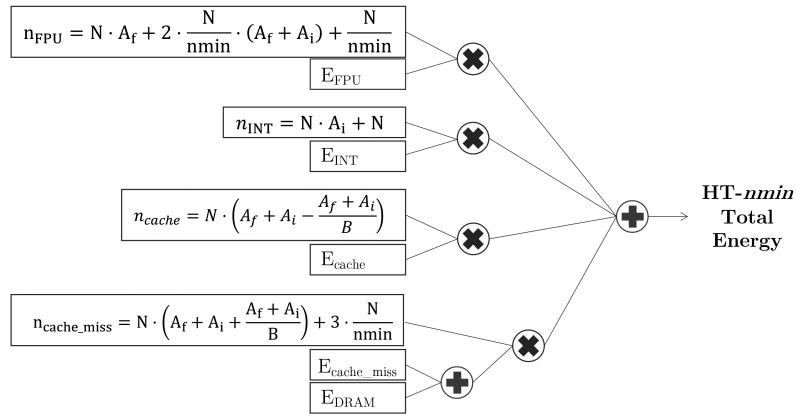


Fig. 4. Energy model for the Hoeffding tree with *nmin* adaptation.

An example of how to apply these steps is shown in Section 4.2.

#### 4.2. Ensembles energy models

The process to create an energy model for any ensemble of other algorithms is summarized in Fig. 3. The goal is to combine the energy model of the ensemble and the energy model of the base algorithm for the ensemble. In the case of Online Bagging, for instance, we need to sum the energy model of Online Bagging plus the energy model of the Hoeffding tree with *nmin* adaptation. If another model is used in the ensemble, it can be substituted by the Hoeffding tree with *nmin* adaptation (dashed box).

This section details, first, the energy model of Hoeffding Trees with *nmin* adaptation (Fig. 4), then the energy models of the ensembles: Online Bagging (Fig. 5), Leveraging Bagging (Fig. 6), Online Boosting (Fig. 7), Online Coordinate Boosting (Fig. 8), Online Accuracy Updated Ensemble (Fig. 9).

##### 4.2.1. Hoeffding trees with *nmin* adaptation

The energy consumption of Hoeffding trees with *nmin* adaptation (HT-*nmin* for the remainder of the paper), explained in detail in Section 3.2, is summarized with the energy model from Fig. 4.



The second step (Fig. 2) after defining the generic energy model is to identify the main functions of the HT-*nmin*. The HT-*nmin* algorithm spends energy on training (traversing the tree, updating the statistics, checking for a split, doing a split), and doing inference (traversing the tree, Naive Bayes or majority class prediction at the leaf). In particular:

- **Traverse the tree:** The energy spent on traversing the tree is calculated as the number of cache misses, which we assume as one access per attribute per instance (for every time as instance is read), as the worst case scenario.
- **Updating statistics:** The energy spent on updating statistics is divided in nominal and numerical attributes. The energy spent in updating attributes regarding computations is one integer computation per instance per nominal attribute, and one floating point operation per numerical attribute. In terms of accesses, for both nominal and numeric attributes, we have a cache miss the first time we access the table, and then one cache hit per nominal attribute, and then a cache miss for every attribute that exceeds the block size.
- **Checking for a split:** To check for a split we need to calculate the entropy for all attributes, calculate the Hoeffding bound, and sort the attributes to obtain the best one. Calculating the entropy, calculating the Hoeffding bound, and sorting the attributes, is one floating point operation per attribute, every *nmin* instances. Regarding memory accesses, we consider one cache miss every *nmin* instances.
- **Doing a split:** To create a new node we consider one floating point operation and one cache miss every *nmin* instances.

Finally, those functions are mapped to the corresponding  $n_{FPU}$ ,  $n_{INT}$ ,  $n_{cache}$ ,  $n_{cache\_miss}$ , as shown in Fig. 4.

What we believe is more important to observe from this energy model is the impact of the *nmin* parameter. The amount of times that the algorithm check for split, which involves a significant part of the energy consumption, depends on the value of *nmin*. That is why having a bad approximation of *nmin* incurs in high energy costs without increasing the predictive performance. This model is the input presented in the energy models of the ensembles from the following paragraphs.

#### 4.2.2. Online Bagging

Online Bagging, presented in Algorithm 4, was proposed by Oza and Russell [36] as a streaming version of traditional ensemble bagging. Bagging is one of the simplest ensemble methods to implement. Non-streaming bagging [9] builds a set of  $M$  base models, training each model with a bootstrap sample of size  $N$  created by drawing random samples with replacement from the original training set. Each base model's training set contains each of the original training examples  $K$  times where  $P(K = k)$  follows a binomial distribution:

$$P(K = k) = \binom{n}{k} p^k (1 - p)^{n-k} = \binom{n}{k} \frac{1}{n} \left(1 - \frac{1}{n}\right)^{n-k}.$$

This binomial distribution for large values of  $n$  tends to a Poisson(1) distribution, where Poisson(1) =  $\exp(-1)/k!$ . Using this fact, [36] proposed *Online Bagging*, an online method that instead of sampling with replacement, gives each example a weight according to Poisson(1).

This algorithm has two main functions, one involved in training the model, that depends on the base algorithm, and the other function that calculates  $k$ . Training the model can be represented as the energy model of the Hoeffding tree with *nmin adaptation*, from Fig. 4. Calculating  $k$  involves one floating point operation per instance ( $N$ ) per model ( $M$ ). The final energy model for Online Bagging is represented in Fig. 5.

**Algorithm 4** Online Bagging(h,d)

---

```

1: for each instance  $d$  do
2:   for each model  $h_m, (m \in 1, 2, \dots, M)$  do
3:      $k = \text{Poisson}(\lambda = 1)$ 
4:     Train the model on the new instance  $k$  times
5:   end for
6: end for

```

---

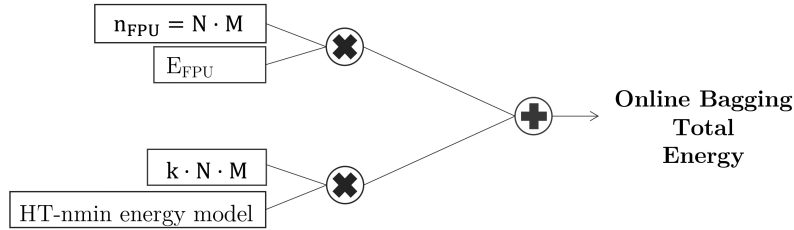


Fig. 5. Online Bagging energy model based on the number of models ( $M$ ) and number of instances ( $N$ ), considering the Hoeffding tree with *nmin* adaptation as the base tree classifier.

#### 4.2.3. Leveraging Bagging

When data is evolving over time, it is important that models adapt to the changes in the stream and evolve over time. ADWIN bagging [7] is the online bagging method of Oza and Russell with the addition of the ADWIN algorithm [3] as a change detector and as an estimator for the weights of the boosting method. When a change is detected, the worst classifier of the ensemble of classifiers is removed and a new classifier is added to the ensemble. *Leveraging Bagging* [6], shown in Algorithm 6, extends ADWIN bagging by leveraging the performance of bagging with two randomization improvements: increasing resampling and using output detection codes. Leveraging bagging increases the weights of this resampling using a larger value  $\lambda$  to compute the value of the Poisson distribution. For every instance, on top of training the model for each ensemble as for Online Bagging, Leveraging Bagging checks if the instance is correctly classified by the model, and inputs such error value to the ADWIN detector (Algorithm 5), to check for a possible change.

**Algorithm 5** ADWIN. **Symbols:**  $h$ : ensemble of models  $h_m$ : model in particular.  $d$ : instance

---

```

1:  $y_{pred} = \text{Classify instance } d \text{ with classifier } h_m$ 
2:  $y = \text{True class of } d$ 
3: if  $y_{pred} == y$  then
4:    $\text{correctlyClassifies} \leftarrow \text{True}$ 
5: end if
6:  $\text{ErrEstim} = \text{getEstimation}(h_m)$ 
7: if  $\text{setInput}(h_m, \text{correctlyClassifies}) == \text{True}$  then
8:   if  $\text{getEstimation}(h_m) > \text{ErrEstim}$  then ▷ After new input from Line 8
9:      $\text{Change} \leftarrow \text{True}$ 
10:  end if
11: end if

```

---

Leveraging Bagging has a similar energy consumption pattern than Online Bagging with the addition of the ADWIN detector and the output codes. The ADWIN detector introduces significant overhead since it has to keep track of the error. For every instance ADWIN calculates if model  $h_m$  correctly classifies instance  $d$ , for each model  $h_m, (m \in 1, 2, \dots, M)$ . It then inputs a 0 for misclassification or 1 for a correct classification to the ADWIN detector. With this information, ADWIN outputs if there has been a

**Algorithm 6** Leveraging Bagging(h,d, $\lambda$ )

---

```

1: for each instance  $d$  do
2:   for each model  $h_m, (m \in 1, 2, \dots, M)$  do
3:      $k = \text{Poisson}(\lambda_d)$ 
4:     Train the model on the new instance  $k$  times
5:     ADWIN( $h, h_m, d$ ) ▷ Call function from Algorithm 5
6:     if ADWIN detects change on  $h_m$  then
7:       Replace classifier with highest error with a new classifier
8:     end if
9:   end for
10: end for

```

---

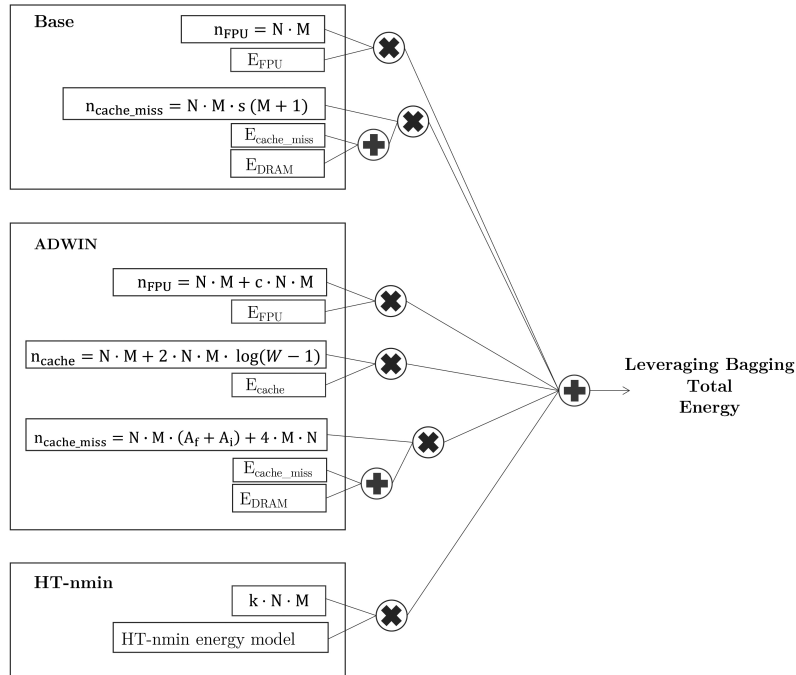


Fig. 6. Energy model for Leveraging Bagging.  $N$  = number of instances,  $M$  = number of models,  $W$  = window size,  $s$  = speed of change,  $c$  = percentage of correctly classified instances.

change in the current window. If the size of the window exceeds the maximum value assigned, ADWIN resizes the window. Finally, if a change is detected, the worst performing classifier is removed from the ensemble, and a new one is created.

The main functions of the baseline algorithm are: calculate  $k$ , train with HT-nmin, and replace classifier. The main functions of ADWIN are: traverse the tree, obtain the true class  $y$ , compare  $y$  with the prediction, get the estimation, and set the input to ADWIN. The following list details these functions in terms of type of operations and number of instances, models, etc. Computing the output codes are omitted since they are not activated by default in the algorithm.

- **Calculate  $k$**  is one FPU operation per instance per model.
- **Replacing the classifier** is one cache miss per classifier to get the error of each classifier, and one cache miss to replace it. That is per instance per model per everytime change is detected ( $S$  variable).
- **Traverse the tree.** As for the HT-nmin, the energy is calculated as the number of cache misses,

- assuming one access per attribute per instance.
- **Obtaining the true class** is one cache miss per instance per model.
  - **Comparing  $y$  with the prediction** is one cache access per instance per model.
  - **Getting the estimation** is just dividing the amount of error by the number of instances, which outputs to one floating point operation per instance per model.
  - **Setting the input** is quite energy consuming, since there are several operations involved. First it inserts the element in the window bucket, which is one cache miss per instance per model. Second it compresses the buckets, which we calculate as one cache miss per instance per model for the first access, and then  $(\log(W) - 1)$  cache accesses per instance per model for the rest of the window accesses,  $W$  being the window size. Third it reduces the window, which is the same energy consumption pattern as compressing the buckets, one cache miss per instance per model and  $(\log(W) - 1)$  cache accesses per instance per model.

Figure 6 shows the energy model of the Leveraging Bagging algorithm, based on the function explanations presented in the previous list.  $M$  is the number of models in the example,  $N$  the number of instances,  $W$  is the window size for ADWIN,  $s$  is the speed of change (the percentage of instances with change),  $c$  is the percentage of correctly classified instances.

#### 4.2.4. Online Boosting

Online Boosting [34,35] is an extension of boosting for streaming data, presented in Algorithm 7. The main difference with Online Bagging is that Online Boosting updates the  $\lambda$  of the Poisson distribution for the next classifier based on the performance of the current classifier (Fig. 4). If classifier  $m$  classifies the instance correctly, that instance will be assigned a lower weight for the next classifier  $m + 1$ . On the other hand, if the instance is classified incorrectly, that instance gets updated with a higher weight. Each instance is passed through each model in sequence.

---

#### Algorithm 7 Online Boosting(h,d)

---

```

1:  $\lambda_d = 1$ 
2: for each instance  $d$  do
3:   for each model  $h_m, (m \in 1, 2, \dots, M)$  do
4:      $k = \text{Poisson}(\lambda_d)$ 
5:     Train the model on the new instance  $k$  times
6:     if instance is correctly classified then
7:       instance weight  $\lambda_d$  updated to a lower value
8:       instance weight  $\lambda_d$  updated to a higher value
9:     end if
10:  end for
11: end for

```

---

Regarding its energy consumption and energy model, we observe that Online Boosting presents a similar energy consumption pattern than Online Bagging. Online Bagging introduces an extra memory access to update the weight, which we assume that the access is a DRAM access because each instance is updated sequentially, not fitting in cache. We assume one instance update per instance per model, the worst case scenario if all instances are classified incorrectly. The detailed energy model is presented in Fig. 7.

#### 4.2.5. Online Coordinate Boosting

Online Coordinate Boosting [37](Algorithm 8) is an extension to Online Boosting, that uses a different procedure to update the weight, to yield a closer approximation to Freund and Schapire's AdaBoost

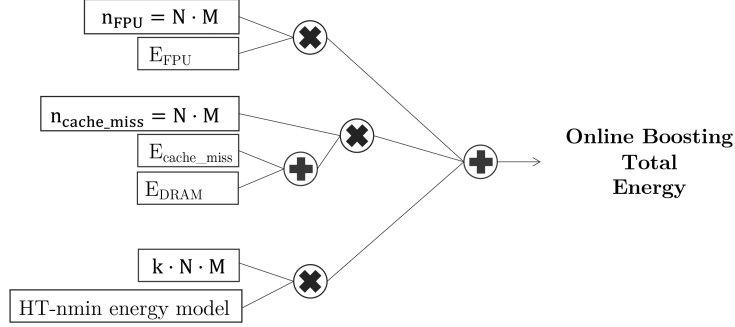


Fig. 7. Online Boosting energy model based on the number of models ( $M$ ) and number of instances ( $N$ ), considering the Hoeffding tree with *nmin adaptation* as the base tree classifier.  $k$  represents the output value of the Poisson distribution from Algorithm 7, line 4.

algorithm [17]. The weight update procedure is derived by minimizing AdaBoost's loss when viewed in an incremental form.

From Algorithm 8 we can observe the different calculations in lines 8, 11, 13, 14, and 15. Once the weight is calculated, the HT-nmin algorithm is trained with weight  $d$ . They use the following formulas to calculate  $\pi_j^+$ ,  $\pi_j^-$ ,  $W_{jk}^+$  and  $W_{jk}^-$ .

---

**Algorithm 8** Online Coordinate Boosting(h,x)
 

---

```

1:  $d = 1$ 
2: for each instance  $x$  do
3:   for each model  $h_m, (m \in 1, 2, \dots, M)$  do
4:     if instance is correctly classified then
5:        $m_j = 1$ 
6:     end if
7:     for every model seen so far  $-1$  do
8:       Calculate  $\pi_j^+$  and  $\pi_j^-$  using Eqs (4) and (5)
9:     end for
10:    for every model seen so far do
11:      Calculate  $W_{jk}^+$  and  $W_{jk}^-$  using Eqs (6) and (7)
12:    end for
13:     $\alpha_j^i = \frac{1}{2} \log \frac{W_{jj}^+}{W_{jj}^-}$ 
14:     $\Delta\alpha_j = \alpha_j^i - \alpha_j^{i-1}$ 
15:     $d \leftarrow de^{-\alpha_j^i m_{ij}}$ 
16:    Train the model on the new instance with weight  $d$ 
17:  end for
18: end for
  
```

---

$$\pi_j^+ = \prod_{k=j_0}^{j-1} \left( \frac{W_{jk}^+}{W_{jj}^+} e^{-\Delta\alpha_k} + \left( 1 - \frac{W_{jk}^+}{W_{jj}^+} \right) e^{\Delta\alpha_k} \right) \quad (4)$$

$$\pi_j^- = \prod_{k=j_0}^{j-1} \left( \frac{W_{jk}^-}{W_{jj}^-} e^{-\Delta\alpha_k} + \left( 1 - \frac{W_{jk}^-}{W_{jj}^-} \right) e^{\Delta\alpha_k} \right) \quad (5)$$

$$W_{jk}^+ = W_{jk}^+ \pi_j^+ + d \mathbf{1}_{[m_{ik}=+1]} \cdot \mathbf{1}_{[m_{ij}=+1]} \quad (6)$$

$$W_{jk}^- = W_{jk}^- \pi_j^- + d \mathbf{1}_{[m_{ik}=-1]} \cdot \mathbf{1}_{[m_{ij}=-1]} \quad (7)$$

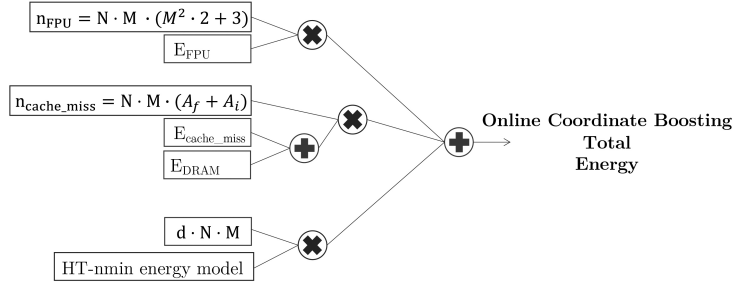


Fig. 8. Energy model for Online Coordinate Boosting.

where  $j$  represents the model of the ensemble,  $j_0$  is set to 0, and  $m_j$  is defined in line 5 of the algorithm. To iterate over the product we introduced the loops in lines 7 and 10.

In relation to energy consumption, this energy model, presented in Fig. 8 adds the extra floating point operations of calculating the values of  $W_{jk}$ ,  $\pi_j$ ,  $\alpha_j^i$ ,  $\Delta\alpha_j$ , and  $d$ , and the cache accesses related to traversing the tree to calculate if the instance is correctly classified. Traversing the tree is the same as for ADWIN, one access per attribute per instance. Calculating  $\pi_j$  can be estimated as

$$\sum_{k=0}^{M-1} k,$$

and  $W_{jk}$  as

$$\sum_{k=0}^M k.$$

Since the sum of  $\pi_j$  and  $W_{jk}$  result in  $M^2$  operations, the amount of floating point operations of lines 8 and 11 can be simplified as:  $M \cdot N(M^2 \cdot 2 + 3)$ . Finally, lines 13, 14, and 15 of Algorithm 8 require one floating point operation per instance per model.

#### 4.2.6. Online Accuracy Updated Ensemble

Batch-incremental methods create models from batches, and they remove them as memory fills up; they cannot learn instance-by-instance. The size of the batch must be chosen to provide a balance between best model accuracy (large batches) and best response to new instances (smaller batches). The *Online Accuracy Updated Ensemble* (OAUE) [11,12], presented in Algorithm 9 maintains a weighted set of component classifiers and predicts the class of incoming examples by aggregating the predictions of components using a weighted voting rule. After processing a new example, each component classifier is weighted according to its accuracy and incrementally trained.

Regarding its energy consumption, we first present the main functions of the algorithm, to then present its energy patterns, combined in an energy model on Fig. 9. The main functions are: create a new classifier, calculate the prediction error, add classifier, weight classifier, and replace classifier.

- **Creating a new classifier** requires one cache miss every  $w$  instances (lines 5 and 12), thus  $\frac{N}{w}$ .
- **Calculating the prediction error** is the same energy as traversing the tree, that as previously explained for the HT-nmin and the Leveraging Bagging algorithms is one cache miss per attribute per instance (line 4), per model.
- **Adding a new classifier** (line 7) is one cache miss every  $w$  instances, only the first  $M$  times.

---

**Algorithm 9** Online Accuracy Updated Ensemble. **Symbols:**  $\varepsilon$ : ensemble  $w$ : window  $M$ : ensemble size
 

---

```

1:  $\varepsilon \leftarrow \emptyset$ 
2:  $C' \leftarrow$  new candidate classifier
3: for each instance  $x^t$  do
4:   Calculate the prediction error of all classifiers  $C_i \in \varepsilon$  on  $x^t$ 
5:   if  $t > 0$  and  $t \bmod w = 0$  then
6:     if  $|\varepsilon| < M$  then
7:       Add classifier  $C'$  to the ensemble  $\varepsilon$ 
8:     else
9:       weight all classifiers  $C_i \in \varepsilon$  and  $C'$  using Eq. (8)
10:      substitute least accurate classifier in  $\varepsilon$  with  $C'$ 
11:    end if
12:     $C' \leftarrow$  new candidate classifier
13:  else
14:    incrementally train classifier  $C'$  with  $x_t$ 
15:    weight all classifiers  $C_i \in \varepsilon$  using Eq. (8)
16:  end if
17:  for all classifiers  $C_i \in \varepsilon$  do
18:    incrementally train classifier  $C_i$  with  $x^t$ 
19:  end for
20: end for
  
```

---

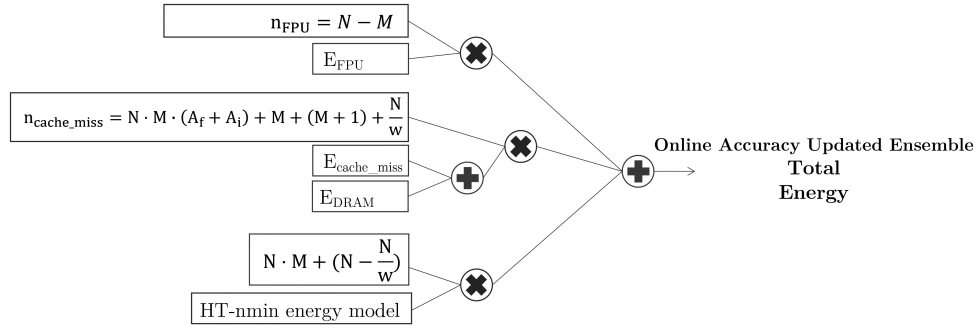


Fig. 9. Energy model for Online Accuracy Updated Ensemble.

- **Calculating the weight** for each classifier requires one floating point operation per classifier per  $N - \frac{N}{w}$  instances, using the Eq. (8) on line 9, and then on line 15 for the rest of the instances. Thus it requires  $N$  floating point operations minus  $M$  (condition on line 6).
- **Replacing the classifier** is the same as the function used in ADWIN, one cache miss per classifier to calculate the error for each classifier, plus one extra cache miss to replace it. This occurs for every  $\frac{N}{w}$  instances.

$$w_i^t = \frac{1}{MSE_r^t + MSE_i^t + \varepsilon} \quad (8)$$

## 5. Experimental design

We have performed several experiments to evaluate our proposed method, *nmin adaptation*, in ensembles of Hoeffding trees. The experiments have a dual purpose: i) first, to see how much energy consumption is reduced by applying the *nmin adaptation* method in ensembles of Hoeffding trees; ii) and second, to see how much accuracy is affected in case of this energy reduction. To fulfill those purposes, we

Table 1  
 Synthetic and real world datasets.  $A_i$  and  $A_f$  represent the number of nominal and numerical attributes, respectively. The details of each dataset is presented in Section 5.1

Dataset	Instances	$A_i$	$A_f$	Class
<b>Synthetic</b>				
RandomTree	1,000,000	5	5	2
Waveform	1,000,000	0	21	3
RandomRBF	1,000,000	0	10	2
LED	1,000,000	24	0	10
Hyperplane	1,000,000	0	10	2
Agrawal	1,000,000	3	6	2
<b>Real World</b>				
Airline	539,383	4	3	2
Electricity	45,312	1	6	2
Poker	829,201	5	5	10
CICIDS	461,802	78	5	6
Forest	581,012	40	10	7

have run a set of five different ensemble algorithms of Hoeffding trees, with and without *nmin adaptation*, and compared their energy consumption and accuracy. We have tested these algorithms over six synthetic datasets and five real world datasets, described in Table 1.

The ensembles used for this paper are: LeveragingBag, OCBBoost, OnlineAccuracyUpdatedEnsemble, OzaBag, and OzaBoost. They have already been described in Section 4. All of them use the Hoeffding tree as the base classifier, with and without the *nmin adaptation* method.

The next subsections explain the datasets in more detail, together with the framework and tools for running the algorithms and measuring the energy consumption.

### 5.1. Datasets

The synthetic datasets have been generated using the already available generators in MOA (Massive Online Analysis) [5]. The real world datasets are publicly available online. Each source is detailed with the explanation of the dataset.

#### 5.1.1. RandomTree

The random tree dataset is inspired in the dataset proposed by the original authors of the Hoeffding tree algorithm [16]. The idea is to build a decision tree, by splitting on random attributes, and assigning random values to the leaves. The new examples are then sorted through the tree and labeled based on the values at the leaves.

#### 5.1.2. Waveform

This dataset is from the UCI repository [15]. A wave is generated as a combination of two or three waves. The task is to predict which one of the three waves the instance represents.

#### 5.1.3. RBF

The radial based function (RBF) dataset is created by selecting a number of centroids, each with a random center, class label and weight. Each new example randomly selects a center, considering that centers with higher weights are more likely to be chosen. The chosen centroid represents the class of the example. More details are given by [8].



Table 2

Accuracy results of running *LeveragingBag*, *OCBoost*, *OnlineAccuracyUpdatedEnsemble (OAUE)*, *OzaBag*, and *OzaBoost* on ensembles of Hoeffding trees, with and without *nmin*. Accuracy is measured as the percentage of correctly classified instances. The *Nmin* and *NoNmin* columns represent running the algorithm with and without *nmin adaptation* respectively. Highest accuracy per dataset and algorithm are presented in bold

Dataset	LeveragingBag		OCBoost		OAUE		OzaBag		OzaBoost	
	nmin	Baseline	nmin	Baseline	nmin	Baseline	nmin	Baseline	nmin	Baseline
Agrawal	<b>94.84</b>	94.79	93.47	<b>93.77</b>	94.97	<b>95.01</b>	94.93	<b>95.09</b>	<b>94.16</b>	93.13
CICIDS	99.63	<b>99.75</b>	48.48	<b>48.53</b>	99.38	<b>99.43</b>	99.38	<b>99.51</b>	<b>99.43</b>	99.38
Hyperplane	<b>90.69</b>	90.46	<b>90.43</b>	89.90	<b>91.96</b>	91.93	<b>90.83</b>	90.68	<b>90.40</b>	90.01
LED	<b>74.02</b>	73.92	<b>17.41</b>	17.32	73.92	<b>73.96</b>	73.89	<b>73.93</b>	<b>73.98</b>	73.81
RandomRBF	95.08	<b>95.30</b>	91.99	<b>92.26</b>	92.17	<b>92.96</b>	93.43	<b>93.99</b>	93.85	<b>93.93</b>
RandomTree	97.28	<b>97.85</b>	93.11	<b>93.28</b>	94.39	<b>94.72</b>	94.47	<b>95.12</b>	95.81	<b>96.77</b>
Waveform	85.77	<b>85.85</b>	55.66	<b>55.85</b>	80.39	<b>80.43</b>	<b>85.52</b>	<b>85.52</b>	84.52	<b>84.53</b>
airline	63.33	<b>63.49</b>	<b>66.09</b>	66.03	68.55	<b>68.69</b>	64.92	<b>64.93</b>	<b>63.46</b>	63.32
Forest	88.90	<b>92.50</b>	73.59	<b>75.44</b>	84.34	<b>85.31</b>	80.89	<b>83.93</b>	87.01	<b>90.71</b>
Elec	89.84	<b>90.44</b>	89.06	<b>91.24</b>	81.54	<b>83.34</b>	81.66	<b>84.08</b>	86.78	<b>87.76</b>
poker	77.54	<b>88.19</b>	75.61	<b>79.87</b>	69.82	<b>71.55</b>	73.92	<b>83.68</b>	85.39	<b>88.40</b>

#### 5.1.4. LED

The attributes of this dataset represent each segment of a digit on a LED display. The goal is to predict which is the digit based on the segments, where there is a 10% chance for each attribute to be inverted [10].

#### 5.1.5. Hyperplane

The hyperplane dataset is generated by creating a set of points that satisfies  $\sum_{i=1}^d w_i x_i = w_0$ , where  $x_i$  is the coordinate for each point [26].

#### 5.1.6. Agrawal

The function generates one of ten different predefined loan functions. The dataset is described in the original paper [1].

#### 5.1.7. airline

The airline dataset [27] predicts if a given flight will be delayed based on attributes such as airport of origin and airline.

#### 5.1.8. Electricity

The electricity dataset is originally described in [22], and is frequently used in the study of performance comparisons. Each instance represents the change of the electricity price based on different attributes such as day of the week, based on the Australian New South Wales Electricity Market.

#### 5.1.9. poker

The poker dataset is a normalized dataset available from the UCI repository. Each instance represents a hand consisting of five playing cards, where each card has two attributes; suit and rank.

#### 5.1.10. Forest

The forest dataset contains the actual forest cover type for a given observation of  $30 \times 30$  meter cells<sup>2</sup>

<sup>2</sup><https://archive.ics.uci.edu/ml/datasets/Coverttype>.

### 5.1.11. CICIDS

The CICIDS dataset is a cybersecurity dataset from 2017 where the task is to detect intrusions [40].

## 5.2. Environment

The framework used to run the algorithms is MOA (Massive Online Analysis). MOA has the most updated data stream mining algorithms, and is widely used in the field. To evaluate the algorithms in terms of accuracy, we have used the *Evaluate Prequential* option, which tests and then trains on subsets of data. This allows to see the accuracy increase in real time.

Measuring energy consumption is a complicated task. There are no simple ways to measure energy, since there are many hardware variables involved. Based on previous work and experience, we believe that the most straight forward approach to measure energy today is by using an interface provided by Intel called RAPL [14]. This interface allows the user to access specific hardware counters available in the processor, which saves energy related measurements of the processor and the DRAM. This approach does not introduce any overhead (in comparison to other approaches such as simulation based), and allows for real time energy measurements. This matches with the requirements in data stream mining. To use the Intel RAPL interface, we use the tool already available: Intel Power Gadget<sup>3</sup>. This tool allows for an energy and power monitoring during the execution of a particular program or script. Since the energy or power is not isolated for that particular program, we have ensured that only our MOA experiments were running on the machine at the time of the measurements. However, since we are aware that there are always programs running in the background, we focus on portraying relative values between different algorithms/methods. For this particular paper, our objective is to understand the difference in energy between using the *nmin* adaptation method and not using the method. Thus, we focus on the difference between those setups, rather than on absolute values of energy.

All the experiments have been run on a machine with an 3.5 GHz Intel Core i7, with 16 GB of RAM, running OSX. We ran the experiments five times and averaged the results.

## 5.3. Reproducibility

The *nmin adaptation* method has been implemented in MOA, to make it openly available once the paper has been reviewed. We have used the implementation in MOA of the ensembles of Hoeffding trees, choosing the Hoeffding tree with *nmin* adaptation as the base classifier. In order to increase the reproducibility of our results, we have made available the code of the *nmin adaptation*, together with the code of all the experiments, the code used to create the tables and plots, and the datasets. It can be obtained in the following link: <https://www.dropbox.com/sh/ahfsp4i99vd1dmc/AABtjEc4EDkfogS1ZMbzl0fea?dl=0>. At the moment is a private repository, but it will be publicly available in GitHub once the paper has been reviewed.

## 6. Results and discussion

The accuracy and energy consumption results of running algorithms LeveragingBag, OCBoost, OnlineAccuracyUpdatedEnsemble, OZaBag, and OzaBag, on the datasets from Table 1, are presented in Tables 2 and 3, respectively.

---

<sup>3</sup><https://software.intel.com/en-us/articles/intel-power-gadget-20>.

Table 3

Energy consumption results of running *LeveragingBag*, *OCBoost*, *OnlineAccuracyUpdatedEnsemble (OAUE)*, *OzaBag*, and *OzaBoost* on ensembles of Hoeffding trees, with and without *nmin*. The energy is measured in joules(J). Lowest energy consumption per dataset and algorithm are presented in bold. The *Nmin* and *NoNmin* columns represent running the algorithm with and without *nmin adaptation* respectively

Dataset	LeveragingBag		OCBoost		OAUE		OzaBag		OzaBoost	
	<i>nmin</i>	Baseline	<i>nmin</i>	Baseline	<i>nmin</i>	Baseline	<i>nmin</i>	Baseline	<i>nmin</i>	Baseline
Agrawal	<b>5121.81</b>	6856.58	<b>1203.68</b>	1522.84	<b>1326.48</b>	1374.64	<b>1236.72</b>	1352.17	<b>2175.24</b>	3018.05
CICIDS	<b>1448.66</b>	1709.49	<b>1511.78</b>	1658.37	<b>1415.23</b>	1528.05	<b>1193.95</b>	1362.44	<b>1463.70</b>	1621.17
Hyperplane	<b>8391.08</b>	14370.83	<b>1009.45</b>	1364.35	<b>856.30</b>	1246.84	<b>1233.10</b>	1815.75	<b>1307.83</b>	1881.81
LED	<b>1854.47</b>	2325.33	<b>1614.94</b>	1865.63	<b>1645.07</b>	1772.10	<b>1218.56</b>	1397.10	<b>1280.74</b>	1507.98
RandomRBF	<b>7389.42</b>	10617.94	<b>1070.32</b>	1347.56	<b>1362.92</b>	1796.12	<b>1614.04</b>	2111.28	<b>1637.68</b>	2293.95
RandomTree	<b>7900.56</b>	12534.89	<b>1510.23</b>	1791.32	<b>1873.98</b>	2452.21	<b>1887.13</b>	2555.93	<b>2465.27</b>	3942.07
Waveform	<b>9164.32</b>	15182.30	<b>1759.72</b>	2298.32	<b>1371.15</b>	1533.26	<b>1544.43</b>	2190.14	<b>1907.51</b>	2593.89
airline	<b>10845.63</b>	12028.00	<b>5015.23</b>	6310.04	<b>1789.47</b>	3004.81	<b>6127.37</b>	7132.72	<b>7057.96</b>	7907.50
Forest	<b>2061.23</b>	2801.53	<b>1619.14</b>	1919.68	<b>1820.17</b>	1969.78	<b>1347.59</b>	2156.80	<b>1693.18</b>	2496.60
Elec	<b>148.72</b>	183.86	<b>93.18</b>	111.81	<b>104.77</b>	121.84	<b>102.15</b>	115.40	<b>95.94</b>	118.15
poker	<b>744.47</b>	1220.16	<b>917.18</b>	1023.73	<b>759.18</b>	844.75	<b>641.70</b>	934.99	<b>801.12</b>	1051.66

Table 4

Difference in accuracy( $\Delta\%$ ) and energy consumption  $\Delta(J)$  between the algorithms running with and without *nmin adaptation*. The difference is measured as the percentage between both approaches. For the  $\Delta(\%)$  column, the higher the value the better, since it means that the *nmin adaptation* approach obtained higher accuracy than the non-*nmin adaptation*. For the  $\Delta(J)$  the lower the better, meaning that we reduced the energy consumption by that percent

Dataset	LeveragingBag		OCBoost		OAUE		OzaBag		OzaBoost	
	$\Delta\%$	$\Delta(J)$	$\Delta\%$	$\Delta(J)$	$\Delta\%$	$\Delta(J)$	$\Delta\%$	$\Delta(J)$	$\Delta\%$	$\Delta(J)$
Agrawal	0.05	-25.30	-0.30	-20.96	-0.04	-3.50	-0.16	-8.54	1.04	-27.93
CICIDS	-0.12	-15.26	-0.05	-8.84	-0.05	-7.38	-0.13	-12.37	0.06	-9.71
Hyperplane	0.23	-41.61	0.53	-26.01	0.03	-31.32	0.15	-32.09	0.39	-30.50
LED	0.11	-20.25	0.09	-13.44	-0.05	-7.17	-0.04	-12.78	0.17	-15.07
RandomRBF	-0.22	-30.41	-0.27	-20.57	-0.79	-24.12	-0.56	-23.55	-0.08	-28.61
RandomTree	-0.56	-36.97	-0.17	-15.69	-0.33	-23.58	-0.65	-26.17	-0.96	-37.46
Waveform	-0.08	-39.64	-0.19	-23.43	-0.03	-10.57	0.00	-29.48	-0.00	-26.46
airline	-0.16	-9.83	0.06	-20.52	-0.14	-40.45	-0.01	-14.09	0.14	-10.74
Forest	-3.60	-26.42	-1.85	-15.66	-0.97	-7.60	-3.04	-37.52	-3.70	-32.18
Elec	-0.60	-19.11	-2.18	-16.67	-1.80	-14.01	-2.42	-11.48	-0.98	-18.80
poker	-10.65	-38.99	-4.26	-10.41	-1.73	-10.13	-9.76	-31.37	-3.01	-23.82
Average	-1.42	-27.62	-0.78	-17.47	-0.54	-16.35	-1.51	-21.77	-0.63	-23.75

Table 4 presents the difference in accuracy and energy consumption between running the algorithm with and without *nmin adaptation*, our proposed method. The column  $\Delta\%$  details the difference, in percentage, between the accuracy of running the algorithm with and without *nmin adaptation*. A negative value represents that the algorithm with *nmin adaptation* obtained lower accuracy than the algorithm with the original implementation. The column  $\Delta(J)$  presents the difference, in percentage, between the energy consumption of running the algorithm with and without *nmin adaptation*. The lower the value the better, since it means that we reduced the energy consumption by that amount.

Looking at Table 4, we can see how for all algorithms and for all datasets *nmin adaptation* reduced the energy consumption by 21 percent, up to a 41 percent. Accuracy is affected by less than 1 percent on average. The poker dataset obtained the highest difference in accuracy between both methods, up to a 10 percent. However, the other datasets show how the original ensembles of Hoeffding trees and the version with *nmin adaptation* are comparable in terms of accuracy.

Figure 10 shows how accuracy increases with the number of instances. Each subplot represents all

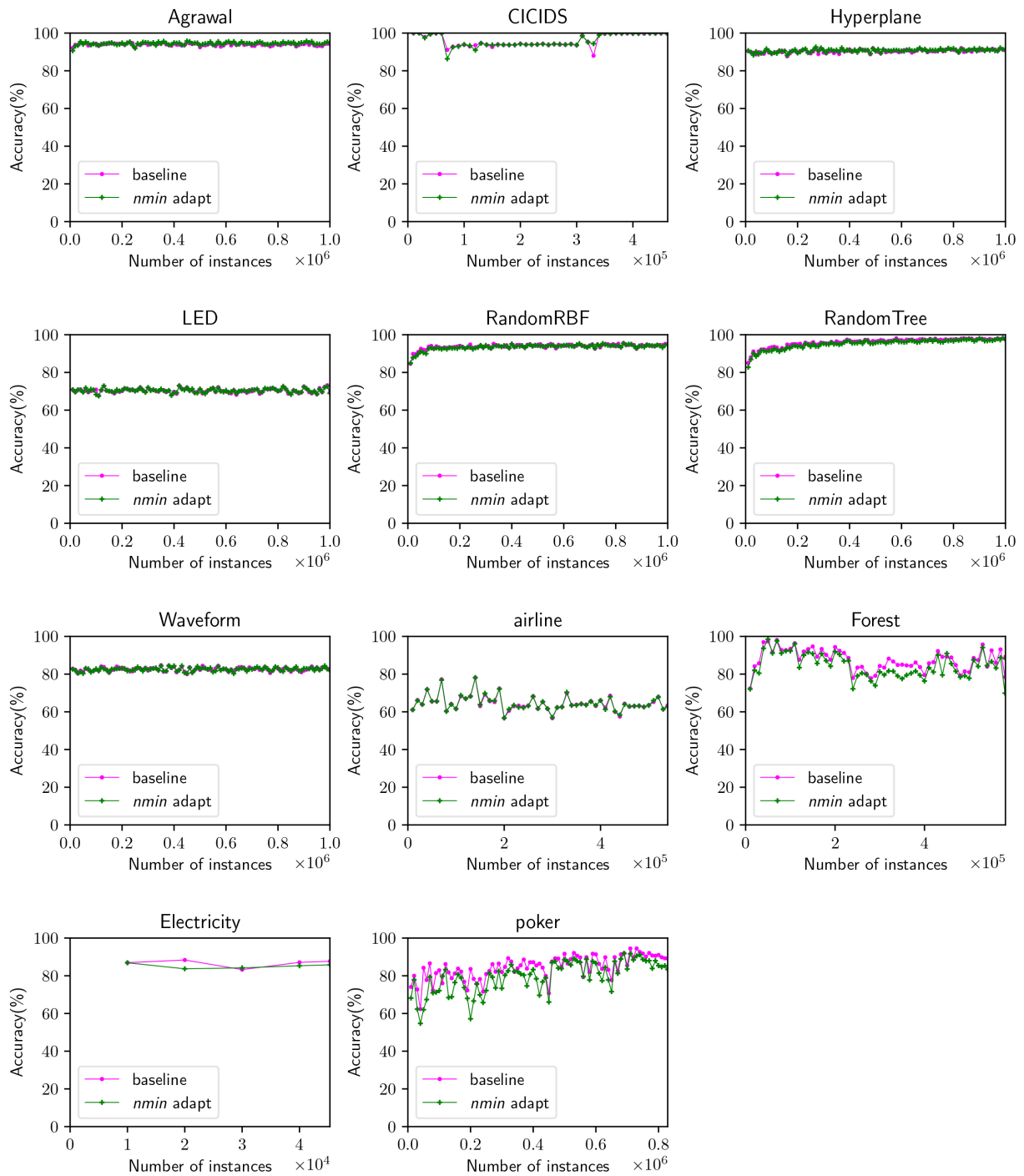


Fig. 10. Accuracy comparison between running the algorithms with and without *nmin* adaptation on all datasets. The results of each dataset are averaged for all algorithms.

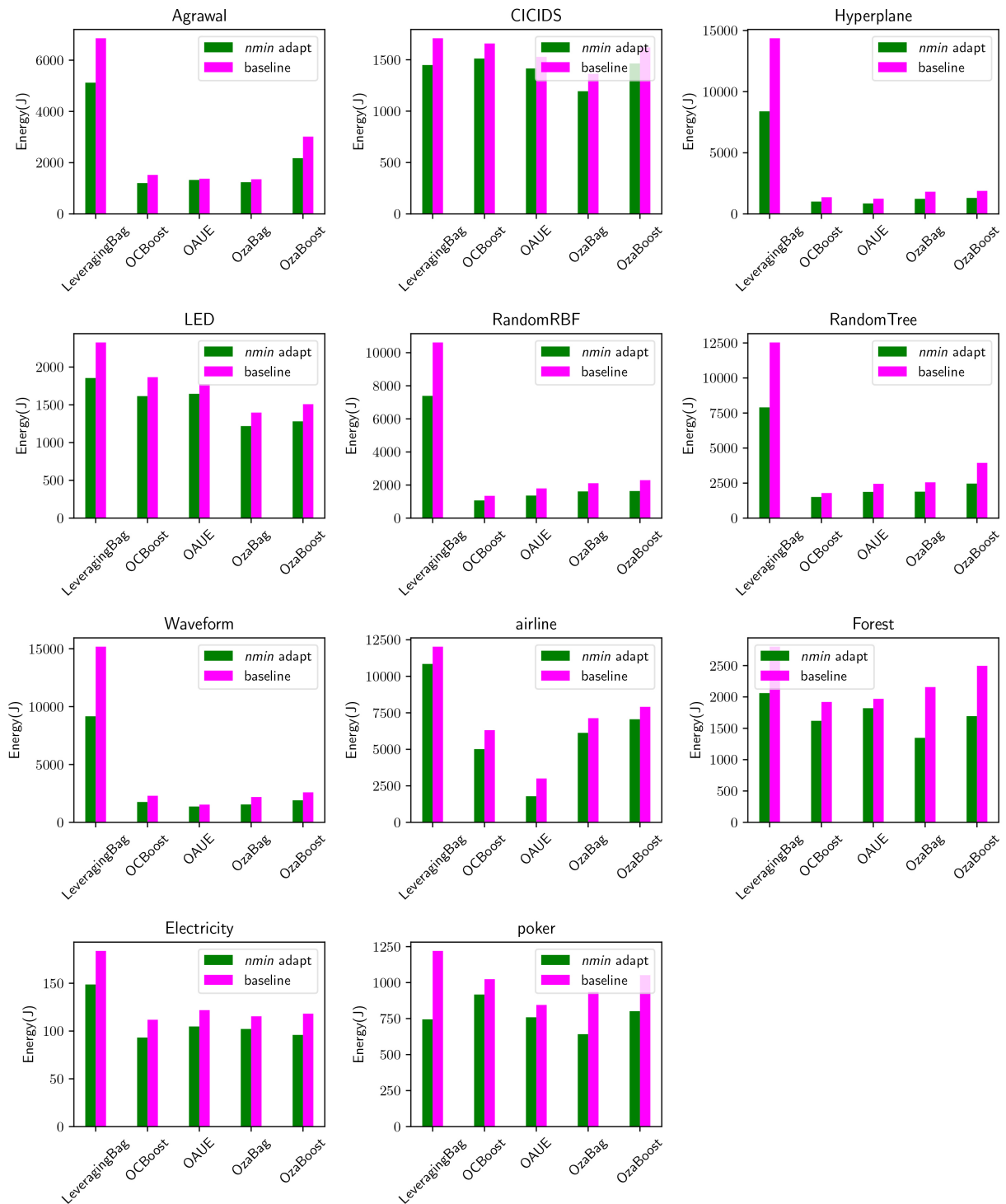


Fig. 11. Energy consumption for all datasets and all algorithms, with and without *nmin adaptation*.

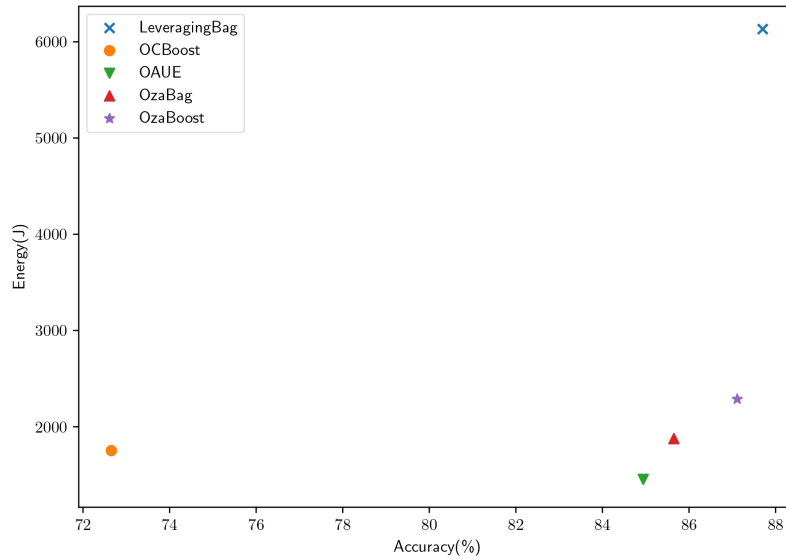


Fig. 12. Energy consumption and accuracy comparison for each algorithm, averaged for all datasets and for the *nmin adaptation* and baseline setups.

algorithms averaged for that particular dataset. Our goal with this figure is to show the difference between running the ensembles with and without *nmin adaptation* in terms of accuracy. We can observe how for most of the datasets (Agrawal, Hyperplane, LED, RandomRBF, RandomTree, Waveform, and airline) the difference is indiscernible, suggesting that the algorithm performs equally well for both the *nmin adaptation* approach and the standard approach. For the poker and forest dataset, we can see how *nmin adaptation* obtains lower accuracy than the standard approach. This was also visible in Table 2.

Figure 11 shows the energy consumption per dataset, per algorithm, with and without *nmin adaptation*. This figure clearly shows how all algorithms with *nmin adaptation* consume less energy than the standard version. This occurs in all datasets, even in datasets where the accuracy is higher for the *nmin adaptation* approach (e.g. Agrawal dataset, OzaBoost algorithm).

At a first glance, and without analyzing the results in depth, one could think that a higher accuracy requires a higher energy cost. However, our results show that there is not a direct relationship between accuracy increase and an increase in energy consumption. This can be observed in Fig. 12, which shows a comparison between accuracy and energy, per algorithm, averaged for all datasets, for the *nmin adaptation* and baseline setups. Looking also at Table 4, for instance at the OzaBoost algorithm, we can see how there is a high energy decrease (30 percent for the Hyperplane dataset) while still obtaining a higher accuracy (0.39 percent). Another algorithm with a high energy decrease is the OAUE for the airline dataset, where we decrease 40% the energy and accuracy is affected by 0.14 percent. A similar energy reduction is obtained in the poker dataset, for the LeveragingBag algorithm. However in this case the accuracy is significantly affected by 10%. These results are very promising, since they suggest that there is not a correlation between energy consumption and accuracy. Thus, there are ways, such as our current approach, to reduce the energy consumption of an algorithm without having to sacrifice accuracy. The importance lies on finding those hotspots that make the algorithm consume energy inefficiently.

In overall, our results show how *nmin adaptation* is able to reduce the energy consumption by 21 percent on average, affecting accuracy by less than one percent on average. This demonstrates that our solution is able to create more energy efficient ensembles of Hoeffding trees trading off less than one percent of accuracy.

## 7. Conclusions

This paper presents a simple, yet effective approach to reduce the energy consumption of ensembles of Hoeffding tree algorithms. This method, *nmin adaptation*, estimates the batch size of instances needed to check for a split, individually for each node. Thus, allowing the Hoeffding tree to grow faster in those branches where the confidence for choosing the best attribute is higher, and delaying the split in those branches where the confidence is lower.

We also present a generic approach to build theoretical energy models for any class of algorithms, together with energy models for the ensembles of Hoeffding trees used in this paper.

We conduct a set of experiments where we compare the accuracy and energy consumption of five ensembles of Hoeffding trees with and without *nmin*, on 11 publicly available datasets. The results show that *nmin adaptation* reduces the energy consumption significantly, on all datasets (up to a 41%, with 21% on average). This is achieved by trading off a few percentages of accuracy (up to a 10% for one particular dataset, on average of less than 1%). Our results also show that there is no clear correlation between a higher accuracy and a higher energy consumption. Thus, opening the possibility for more research to create more energy efficient algorithms without sacrificing accuracy.

While data stream mining approaches are focused on running on edge devices, still there has not been much research that tackle the energy efficiency of the approaches, which is a key variable for these devices. This study provides one step forward into achieving more energy efficient algorithms that are able to run in embedded devices. For future work, we plan on designing more energy efficient Hoeffding trees that are able to compete with high accurate current approaches [32].

## References

- [1] R. Agrawal, T. Imielinski and A. Swami, Database mining: A performance perspective, *IEEE Transactions on Knowledge and Data Engineering* **5**(6) (1993), 914–925.
- [2] Y. Ben-Haim and E. Tom-Tov, A streaming parallel decision tree algorithm, *Journal of Machine Learning Research* **11** (Feb 2010), 849–872.
- [3] A. Bifet and R. Gavaldà, Learning from time-changing data with adaptive windowing, In *Proceedings of the 2007 SIAM international conference on data mining*, SIAM, 2007, pp. 443–448.
- [4] A. Bifet and R. Gavaldà, Adaptive learning from evolving data streams, In *International Symposium on Intelligent Data Analysis*, Springer, 2009, pages 249–260.
- [5] A. Bifet, G. Holmes, R. Kirkby and B. Pfahringer, Moa: Massive online analysis, *Journal of Machine Learning Research* **11** (May 2010), 1601–1604.
- [6] A. Bifet, G. Holmes and B. Pfahringer, Leveraging bagging for evolving data streams, In *Joint European conference on machine learning and knowledge discovery in databases*, Springer, 2010, pp. 135–150.
- [7] A. Bifet, G. Holmes, B. Pfahringer, R. Kirkby and R. Gavaldà, New ensemble methods for evolving data streams, In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Paris, France*, June 28–July 1 2009, 2009, pp. 139–148.
- [8] A. Bifet, J. Zhang, W. Fan, C. He, J. Zhang, J. Qian, G. Holmes and B. Pfahringer, Extremely fast decision tree mining for evolving data streams, In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ACM, 2017, pp. 1733–1742.
- [9] L. Breiman, Bagging predictors, *Machine Learning* **24**(2) (1996), 123–140.
- [10] L. Breiman, *Classification and regression trees*, Routledge, 2017.
- [11] D. Brzeziński and J. Stefanowski, Accuracy updated ensemble for data streams with concept drift, In *International conference on hybrid artificial intelligence systems*, Springer, 2011, pp. 155–163.
- [12] D. Brzeziński and J. Stefanowski, Combining block-based and online methods in learning ensembles from concept drifting data streams, *Information Sciences* **265** (2014), 50–67.
- [13] Y.-H. Chen, T. Krishna, J.S. Emer and V. Sze, Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks, *IEEE Journal of Solid-State Circuits* **52**(1) (2017), 127–138.

- [14] H. David, E. Gorbatov, U.R. Hanebutte, R. Khanna and C. Le, Rapl: memory power estimation and capping, In *Proceedings of the 16th ACM/IEEE international symposium on Low power electronics and design*, 2010, pp. 189–194.
- [15] D. Dheeru and E. Karra Taniskidou, UCI machine learning repository, 2017.
- [16] P. Domingos and G. Hulten, Mining high-speed data streams, In *Proc. 6th SIGKDD International Conference on Knowledge discovery and data mining*, 2000, pp. 71–80.
- [17] Y. Freund and R.E. Schapire, A decision-theoretic generalization of on-line learning and an application to boosting, In *Computational Learning Theory, Second European Conference, EuroCOLT '95, Barcelona, Spain, March 13–15 1995, Proceedings*, 1995, pp. 23–37.
- [18] E. García-Martín, N. Lavesson, H. Grahn, E. Casalicchio and V. Boeva, Hoeffding trees with nmin adaptation, In *2018 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*, 2018.
- [19] K. Gauen, R. Rangan, A. Mohan, Y.-H. Lu, W. Liu and A.C. Berg, Low-power image recognition challenge, In *Design Automation Conference (ASP-DAC), 2017 22nd Asia and South Pacific*, IEEE, 2017, pp. 99–104.
- [20] S. Han, H. Mao and W.J. Dally, Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding, In *International Conference on Learning Representations (ICLR)*, 2016.
- [21] S. Han, J. Pool, J. Tran and W. Dally, Learning both weights and connections for efficient neural network, In *Advances in neural information processing systems*, 2015, pp. 1135–1143.
- [22] M. Harries, Splice-2 comparative evaluation: Electricity pricing, Technical report, The University of New South Wales, 1999.
- [23] J.L. Hennessy and D.A. Patterson, *Computer architecture: a quantitative approach*, Elsevier, 2011.
- [24] W. Hoeffding, Probability inequalities for sums of bounded random variables, *Journal of the American Statistical Association* **58**(301) (1963), 13–30.
- [25] M. Horowitz, 1.1 computing's energy problem (and what we can do about it), In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2014 IEEE International*, IEEE, 2014, pp. 10–14.
- [26] G. Hulten, L. Spencer and P. Domingos, Mining time-changing data streams, In *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, ACM, 2001, pp. 97–106.
- [27] E. Ikonomovska, Datasets, Retrieved from [http://kt.ijs.si/elena\\_ikonomovska/data.html](http://kt.ijs.si/elena_ikonomovska/data.html), 2013. Online; accessed 1 August 2018.
- [28] J.G. Koomey, S. Berard, M. Sanchez and H. Wong, Assessing trends in the electrical efficiency of computation over time, *IEEE Annals of the History of Computing* **17** (2009).
- [29] N. Kourtellis, G.D.F. Morales, A. Bifet and A. Murdopo, Vht: Vertical hoeffding tree, In *Big Data (Big Data), 2016 IEEE International Conference on*, IEEE, 2016, pp. 915–922.
- [30] V. Losing, B. Hammer and H. Wersing, KNN classifier with self adjusting memory for heterogeneous concept drift, In *IEEE 16th International Conference on Data Mining (ICDM)*, Dec 2016, pp. 291–300.
- [31] V. Losing, H. Wersing and B. Hammer, Enhancing very fast decision trees with local split-time predictions, In *2018 IEEE International Conference on Data Mining (ICDM)*, IEEE, 2018, pp. 287–296.
- [32] C. Manapragada, G. Webb and M. Salehi, Extremely fast decision tree, In *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, ACM, 2018.
- [33] D. Marrón, E. Ayguadé, J.R. Herrero, J. Read and A. Bifet, Low-latency multi-threaded ensemble learning for dynamic big data streams, In *Big Data (Big Data), 2017 IEEE International Conference on*, IEEE, 2017, pp. 223–232.
- [34] N.C. Oza, Online bagging and boosting, In *2005 IEEE international conference on systems, man and cybernetics*, volume 3, Ieee, 2005, pp. 2340–2345.
- [35] N.C. Oza and S. Russell, Experimental comparisons of online and batch versions of bagging and boosting, In *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, ACM, 2001, pp. 359–364.
- [36] N.C. Oza and S. Russell, Online bagging and boosting, In *In Artificial Intelligence and Statistics 2001*, Morgan Kaufmann, 2001, pp. 105–112.
- [37] R. Pelossof, M. Jones, I. Vovsha and C. Rudin, Online coordinate boosting, In *Computer Vision Workshops (ICCV Workshops), 2009 IEEE 12th International Conference on*, IEEE, 2009, pp. 1354–1361.
- [38] C.F. Rodrigues, G. Riley and M. Luján, Fine-grained energy profiling for deep convolutional neural networks on the jetson tx1, In *Workload Characterization (IISWC), 2017 IEEE International Symposium on*, IEEE, 2017, pp. 114–115.
- [39] C.F. Rodrigues, G. Riley and M. Luján, Synergy: An energy measurement and prediction framework for convolutional neural networks on jetson tx1, In *International Conference on Parallel and Distributed Processing Techniques and Applications*, CSREA Press, 2018, pp. 375–382.
- [40] I. Sharafaldin, A.H. Lashkari and A.A. Ghorbani, Toward generating a new intrusion detection dataset and intrusion traffic characterization, In *ICISSP*, 2018, pp. 108–116.