Adaptive XGBoost for Evolving Data Streams

Jacob Montiel*, Rory Mitchell*, Eibe Frank*, Bernhard Pfahringer*, Talel Abdessalem† and Albert Bifet*†
* Department of Computer Science, University of Waikato, Hamilton, New Zealand
Email: {jmontiel, eibe, bernhard, abifet}@waikato.ac.nz, r.a.mitchell.nz@gmail.com
† LTCI, Télécom ParisTech, Institut Polytechnique de Paris, Paris, France
Email: talel.abdessalem@telecom-paris.fr

***Abstract*—Boosting is an ensemble method that combines base models in a sequential manner to achieve high predictive accuracy. A popular learning algorithm based on this ensemble method is *eXtreme Gradient Boosting* (XGB). We present an adaptation of XGB for classification of evolving data streams. In this setting, new data arrives over time and the relationship between the class and the features may change in the process, thus exhibiting concept drift. The proposed method creates new members of the ensemble from mini-batches of data as new data becomes available. The maximum ensemble size is fixed, but learning does not stop when this size is reached because the ensemble is updated on new data to ensure consistency with the current concept. We also explore the use of concept drift detection to trigger a mechanism to update the ensemble. We test our method on real and synthetic data with concept drift and compare it against batch-incremental and instance-incremental classification methods for data streams.**

***Index Terms*—Ensembles, Boosting, Stream Learning, Classification**

I. Introduction

The eXtreme Gradient Boosting (XGB) algorithm is a popular method for supervised learning tasks. XGB is an ensemble learner based on boosting that is generally used with decision trees as weak learners. During training, new weak learners are added to the ensemble in order to minimize the objective function. Different to other boosting techniques, the complexity of the trees is also considered when adding weak learners: trees with lower complexity are preferred. Although configuring the multiple hyper-parameters in XGB can be challenging, it performs at the state-of-the-art if this is done properly.

An emerging approach to machine learning comes in the form of learning from evolving data streams. It provides an attractive alternative to traditional batch learning in multiple scenarios. An example is fraud detection for online banking operations, where training is performed on massive amounts of data. In this case, consideration of runtime is critical: waiting for a long time until the model is trained means that potential frauds may pass undetected. Another example is the analysis of communication logs for security, where storing all logs is impractical (and in most cases unnecessary). The requirement to store all data is a significant limitation of methods that need to perform multiple passes over the data.

Stream learning comprises a set of additional challenges, such as: models have access to the data only once and need to process it on the go since new data arrives continuously; models need to provide predictions at any moment in time; and there is a potential change in the relationship between features and learning targets, known as concept drift. Concept drift is a challenging problem, and is common in many real-world applications that aim to model dynamic systems. Without proper intervention, batch methods will fail after a concept drift because they are essentially trained for a different problem (concept). A common approach to deal with this phenomenon, usually signaled by the degradation of a batch model, is to replace the model with a new model, which implies a considerable investment on resources to collect and process data, train new models and validate them. In contrast, stream models are continuously updated and adapt to the new concept.

We list the contributions of our work as follows:

- We propose an adaptation of the eXtreme Gradient Boosting algorithm for evolving data streams.
- We provide an open-source implementation[1] of the proposed algorithm.
- We perform a thorough evaluation of the proposed method in terms of performance, hyper-parameter relevance, memory, and training time.
- Our experimental results update the existing literature comparing instance-incremental and batch-incremental methods, with current state-of-the-art methods.

This paper is organized as follows: In Section II we examine related work. The proposed method is introduced in Section III. Section IV describes the methodology for our experiments. Results are discussed in Section V. We present our conclusions in Section VI.

II. Related Work

Ensemble methods are a popular approach to improve predictive performance and robustness. One of the first techniques to address concept drift with ensembles trained on streaming data is the *SEA* algorithm [1], a variant of bagging [2] which maintains a fixed-size ensemble trained incrementally on chunks of data. *Online Bagging* [3] is an adaptation of bagging for data streams. Similar to batch bagging, M models are generated and then trained on N samples. Different to batch bagging, where samples are selected with replacement, in Online Bagging, samples are assigned a weight based on $Poisson(1)$. *Leveraging Bagging* [4] builds upon Online Bagging. The key idea is to increase accuracy and diversity

[1] https://github.com/jacobmontiel/AdaptiveXGBoostClassifier

on the ensemble via randomization. Additionally, Leveraging Bagging uses the ADWIN [5] drift detector; if a change is detected, the worst member in the ensemble is replaced by a new one. The *Ensemble of Restricted Hoeffding Trees* [6] combines the predictions of multiple tree models built from subsets of the full feature space using stacking. The *Self Adjusting Memory* algorithm [7] builds an ensemble with models targeting current or former concepts. SAM works under the *Long-Term — Short-Term* memory model (LTM-STM), where the STM focuses on the current concept and the LTM retains information about past concepts. *Adaptive Random Forest* [8] is an adaptation of the Random Forest method designed to work on data streams. The base learners are Hoeffding Trees, attributes are randomly selected during training, and concept drift is tracked using ADWIN on each member of the ensemble.

In the batch setting, boosting is an extremely popular ensemble learning strategy. The *Pasting Votes* [9] method is the first to apply boosting on large data sets by using different sub-sets of data for each boosting iteration; it does not require to store all data and potentially can be used on stream data. A similar approach is *Racing Committees* [10]. Different to [9], Racing Committees includes a adaptive pruning strategy to manage resources (time and memory). In the stream setting, a number of approaches for boosting have been proposed. *Learn++.NSE* [11], inspired in *AdaBoost* [12], generates a batch-based ensemble of weak classifiers trained on different sample distributions and combines weak hypotheses through weighted majority voting.

In stream learning, two main branches of algorithms can be distinguished depending on the schema used to train a model. *Instance-incremental* methods [3], [4], [6], [8], [7], where a single sample is used at a time, and *batch-incremental* methods [9], [11], [10] that use batches of data: Once a given number of samples are stored in the batch, they are used to train the model. The *Accuracy-Weighted Ensembles* [13], is a framework for mining streams with concept drift using weighted classifiers. Members of the ensemble are discarded by an instance-based pruning strategy if they are below a confidence threshold. A relevant study is [14], where the authors compare batch-incremental and instance-incremental methods for the task of classification of evolving data streams. While instance-incremental methods perform better on average, batch-incremental methods achieve similar performance in some scenarios.

III. Adapting XGB for Stream Learning

In this section, we present an adaptation of the XGB algorithm [15] suitable for evolving data streams.

A. Preliminaries

The goal of supervised learning is to predict the responses $Y = \{y_i\} : i \in \{1, 2, \ldots, n\}$ corresponding to a set of feature vectors $X = \{\vec{x}_i\} : i \in \{1, 2, \ldots, n\}$. Ensemble methods yield predictions $\hat{y}_i$ corresponding to a given input $\vec{x}_i$ by combining the predictions of all the members of the ensemble E. In this paper, we focus on binary classification, that is, $y \in \{C_1, C_2\}$.

In the case of boosting, the ensemble E is created sequentially. In each iteration k, a new base function f_k is selected and added to the ensemble so that the loss ℓ of the ensemble is minimized:

$$\ell(\mathrm{E}) = \sum_{k=1}^{K} \ell(Y, \hat{Y}^{(k-1)} + f_k(X)) + \Omega(f_k). \tag{1}$$

Here, K is the number of ensemble members and each $f_k \in \mathcal{F}$ with $\mathcal{F}$ being the space of possible base functions. Commonly, this is the space of regression trees, so each base function is a step-wise constant predictor and the ensemble prediction $\hat{Y}$, which is simply the sum of all K base functions, is also step-wise constant. The regularization parameter Ω penalizes complex functions.

The ensemble is created using forward additive modeling, where new trees are added one at a time. At step k, the training data is evaluated on existing members of the ensemble and the corresponding prediction scores $Y^{(k)}$ are used to drive the creation of new members of the ensemble. The base functions predictions are combined additively:

$$\hat{Y}^{(k)} = \sum_{k=1}^{K} f_k(X) = \hat{Y}^{(k-1)} + f_k(X) \tag{2}$$

The final prediction for a sample $\hat{y}_i$ is the sum of the predictions for each tree f_k in the ensemble.

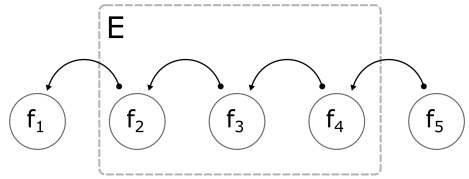
$$\hat{y}_i = \sum_{k=1}^{K} f_t(x_i) \tag{3}$$

B. Adaptive eXtreme Gradient Boosting

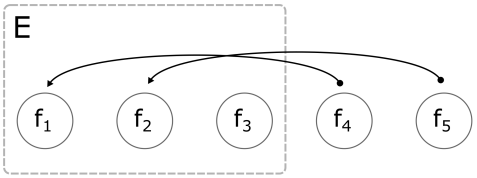
In the batch setting, XGB training is performed using all available data (X, Y). However, in the stream setting, we receive new data over time, and this data may be subject to concept drift. A continuous stream of data can be defined as $A = \{(\vec{x}_t, y_t)\} | t = 1, \ldots, T$ where $T \to \infty$, $\vec{x}_t$ is a feature vector, and y_t the corresponding target. We now describe a modified version of the XGB algorithm for this scenario, called ADAPTIVE EXTREME GRADIENT BOOSTING (AXGB). AXGB uses an incremental strategy to create the ensemble. Instead of using the same data to select each base function f_i, it uses sub-samples of data obtained from non-overlapping (tumbling) windows. More specifically, as new data samples arrive, they are stored in a buffer $w = (\vec{x}_i, y_i) : i \in \{1, 2, \ldots, W\}$ with size $|w| = W$ samples. Once the buffer is full, AXGB proceeds to train a single f_k. We can rewrite Eq. 2 as:

$$\hat{Y}^{(k)} = \sum_{k=1}^{K} f_k(w_k) = \hat{Y}^{(k-1)} + f_k(w_k) \tag{4}$$

The index k of the new base function within the ensemble determines the way in which this function is obtained. If it



(a) Push



(b) Replace

Fig. 1: Ensemble creation strategies.

is the first member of the ensemble, f_1, then the data in the buffer is used directly. If $k > 1$, then the data is passed through the ensemble and the residuals from the first $k-1$ models in the ensemble are used to obtain the new base function f_k.

C. Updating the Ensemble

Given that data streams are potentially infinite and may change over time, learned predictors must be updated continuously. Thus, it is essential to define a strategy to keep the AXGB ensemble updated once it is full. In the following, we consider two strategies for this purpose:

- A *push* strategy ($\text{AXGB}_{[p]}$), shown in Figure 1a, where the ensemble resembles a queue. When new models are created they are appended to the ensemble. If the ensemble is full then older models are removed before appending a new model.
- A *replacement* strategy ($\text{AXGB}_{[r]}$), shown in Figure 1b, where older members of the ensemble are replaced with newer ones.

Notice that in both cases, we have to wait K iterations to have a completely new ensemble. However, in $\text{AXGB}_{[r]}$, newer models have a more significant impact on predictions than older ones, while the reverse is true for $\text{AXGB}_{[p]}$.

A requirement in stream learning is that models are ready to provide predictions at any time. Given the incremental nature of AXGB, if the window (buffer) size W is fixed, the ensemble will require $K \cdot W$ samples to create the full ensemble. A negative aspect of this approach is that performance can be sub-optimal at the beginning of the stream. To overcome this, AXGB uses a dynamic window size W that doubles in each iteration from a given minimum size W_{min} until a maximum size W_{max} is reached. In other words, it grows exponentially until reaching W_{max}. The window size, $W(i)$, for the ith iteration is defined as:

$$W(i) = min(W_{min} \cdot 2^i, W_{max}) \tag{5}$$

From Eq. 5, we see that the number of iterations i required to reach the maximum window size is:

$$i = \left\lceil \log_2 \left(\frac{W_{max}}{W_{min}} \right) \right\rceil \tag{6}$$

Similarly, we see that the number of samples required to create K models to fill the ensemble is smaller when using the dynamic window size approach than when using a fixed window size W_{max} given that

$$\sum_{i=0}^{K-1} W_{min} \cdot 2^i \ll K \cdot W_{max}. \tag{7}$$

Because we monotonically increase the window size, we see that both ensemble update strategies replace base functions trained on small windows with newer ones trained on more data.

D. Handling Concept Drift

Although the incremental strategy used by AXGB to create the ensemble indirectly deals with concept drift—new members of the ensemble are added based on newer data—it may be too slow to adjust to fast drifts. Hence, we use ADWIN [5], to track changes in the performance of AXGB, as measured by a metric such as classification accuracy. We use subscript $_A$ to denote ADWIN, therefore the concept-drift-aware version of AXGB is referred in the following as AXGB_A.

AXGB_A uses the change detection signal obtained from ADWIN to trigger a mechanism to update the ensemble. This mechanism works as follows:

1) Reset the size of the window w to the defined minimum size W_{min}.
2) Train and add new members to the ensemble depending on the chosen strategy:
   a) Push: New ensemble members are appended to the ensemble while the oldest are removed from it. Since new models are trained on increasing window sizes they will be added at a faster rate initially; this effectively works as a *flushing* strategy to update the ensemble.
   b) Replacement: The index used to replace old members of the ensemble is reset so that it points to the beginning of the ensemble. There are two considerations: First, new models replace the oldest ones in the ensemble. Second, new models are trained without considering the residuals of old models that were trained on the older concept.

IV. Experimental Methodology

In this section, we describe the methodology of our tests, which we classify into the following categories: predictive performance, hyper-parameter relevance, memory usage / model complexity, and training time.

1) **Predictive performance**. Our first set of tests evaluates the predictive performance of AXGB. For this we use both, synthetic and real-world data sets. We then proceed to compare AXGB against other learning methods. This

comparison is defined by the nature of the learning method as follows:

a) *Batch-incremental methods*. In this type of learning methods, batches of samples are used to incrementally update the model. We compare AXGB against a batch-incremental ensemble created by combining multiple per-batch base models. New base models are trained independently on disjoint batches of data (windows). When the ensemble is full, older models are replaced with newer ones. Predictions are formed by majority vote. In order to compare this approach with AXGB, we use XGB as the base batch-learner to learn an ensemble for each batch. Thus, our batch-incremental model is an ensemble of XGB ensembles. We refer to this batch-incremental method as BXGB. We also consider Accuracy-Weighted Ensembles with Decision Trees as the base batch-learner. We refer to this method as AWE-J48. We choose this configuration since AWE-J48 is reported as the top batch-incremental performer in [14], so it serves as a baseline for batch-incremental methods.

b) *Instance-incremental methods*. We are also interested in comparing AXGB against methods that update their model one instance at a time. The following instance-incremental methods are used in our tests: Adaptive Random Forest (ARF), Hoeffding Adaptive Tree (HAT), Leverage Bagging with Hoeffding Tree as base learner (LB_{HT}), Oza Bagging with Hoeffding Tree as base learner (OB_{HT}), Self Adjusting Memory with kNN (SAM_{kNN}) and the Ensemble of Restricted Hoeffding Trees (RHT). In [14], LB_{HT} is reported as the top instance-incremental performer.

We perform non-parametric tests to verify whether there are statistically significant differences between algorithms, as described in [16], [17].

2) **Hyper-parameter relevance**. The XGB algorithm relies on multiple hyper-parameters, which can make the model hard to tune for different problems. We are interested in analyzing the impact of hyper-parameters in AXGB. For this purpose, we use a hyper parameter tuning setup where a model is trained on the first 30% of the data stream using different combinations of hyper-parameters. Then, the best performers during the training phase are evaluated on the remaining 70% of the stream. To evaluate the influence of hyper-parameters, we compare performance between AXGB and BXGB.

3) **Memory usage and model complexity**. The potentially infinite number of samples in data streams requires resources such as time and memory to be properly managed. We use the total number of nodes in the ensemble to gain insight into memory usage and model complexity as AXGB is trained on a data stream. We compare the proposed versions of AXGB against a baseline XGB model trained on all the data from the stream. The baseline number of nodes in the XGB model is expected to be larger than the number of nodes in incremental-models that evolve with the stream. By analyzing memory usage and model complexity we aim to get intuition on the evolution of the model over time.

4) **Training time**. Another relevant way to analyze the proposed method is in terms of training time. We compare the training time of the different versions of AXGB against XGB, reporting results in terms of training time (seconds) and in terms of throughput (samples per second).

Our implementation of AXGB is based on the official XGB C-API[2] on top of scikit-multiflow[3] [18], a stream learning framework in Python. Tests are performed using the official XGB implementation, the implementations of ARF, RHT and AWE in MOA [19], and for the rest of the methods, the implementations available in scikit-multiflow. Default parameters of the algorithms are used unless otherwise specified.

A. Data

In the following, we provide a short description of the synthetic and real world datasets used in our tests. All datasets are publicly available. A summary of the datasets used in our experiments is available in Table I.

- *AGRAWAL* – Based on the Agrawal generator [20], represents a data stream with six nominal and three numerical features. Different functions map instances into two different classes. Three abrupt drifts are simulated for AGR_a and three gradual drifts for AGR_g.
- *HYPER* – A stream with fast incremental drifts where a d-dimensional hyperplane changes position and orientation. Obtained from a random hyperplane generator [21].
- *SEA* – A data stream with three numerical features where only two attributes are related to the target class. Created using the SEA generator [1]. Three abrupt drifts are simulated for SEA_a and three gradual drifts for SEA_g.
- *AIRLINES* – Real world data containing information from scheduled departures of commercial flights within the US. The objective is to predict if a flight will be delayed.
- *ELECTRICITY* – Data from the Australian New South Wales Electricity Market, where prices are not fixed but change based on supply and demand. The two target classes represent changes in the price (up or down).
- *WEATHER* – Contains weather information collected between 1949–1999 in Bellevue, Nebraska. The goal is to predict rain on a given date.

V. Experimental Results

The results discussed in this section provide information about predictive performance, parameter relevance, memory and time for the different versions of AXGB.

[2] https://github.com/dmlc/xgboost
[3] https://github.com/scikit-multiflow/scikit-multiflow

TABLE I: Datasets. [Type] S: synthetic data; R: real world data. [Drifts] A: abrupt, G: gradual; I_f: incremental fast, ?: drifts with unknown nature.

Dataset	# instances	# features	# classes	Type	Drift
AGR_a	1000000	9	2	S	A
AGR_g	1000000	9	2	S	G
$HYPER_f$	1000000	10	2	S	I_f
SEA_a	1000000	3	2	S	A
SEA_g	1000000	3	2	S	G
AIRL	539383	7	2	R	?
ELEC	45312	6	2	R	?
WEATHER	18159	8	2	R	?

TABLE II: Parameters used for batch-incremental methods.

Parameter	AXGB *	BXGB	AWE-J48
ensemble size	30	30	30
ensemble size (base learner)	-	30	-
max window size	1000	1000	1000
min window size	1	-	-
max depth	6	6	-
learning rate	0.3	0.3	-

* The same parameter configuration is used for all variations: $AXGB_{[p]}$, $AXGB_{A[p]}$, $AXGB_{[r]}$ and $AXGB_{A[r]}$.

TABLE IV: Comparing performance of AXGB vs instance-incremental methods.

Dataset	$AXGB_{[r]}$	$AXGB_{A[r]}$	ARF	HAT	LB_{HT}	OB_{HT}	SAM_{kNN}	RHT
AGR_a	0.931	0.928	**0.939**	0.807	0.881	0.915	0.686	0.936
AGR_g	0.907	0.901	**0.912**	0.792	0.858	0.847	0.669	0.911
AIRL	0.621	0.618	**0.680**	0.608	0.670	0.658	0.605	0.648
ELEC	0.739	0.747	0.855	**0.874**	0.836	0.794	0.799	0.873
$HYPER_f$	0.847	0.847	0.849	0.869	0.814	0.806	0.870	**0.896**
SEA_a	0.875	0.874	**0.897**	0.827	0.891	0.869	0.876	0.889
SEA_g	0.873	0.872	**0.893**	0.825	0.889	0.869	0.873	0.885
WEATHER	0.774	0.747	**0.791**	0.693	0.783	0.749	0.781	0.758
avg. rank	4.750	5.688	**1.625**	6.125	3.750	6.000	5.313	2.750

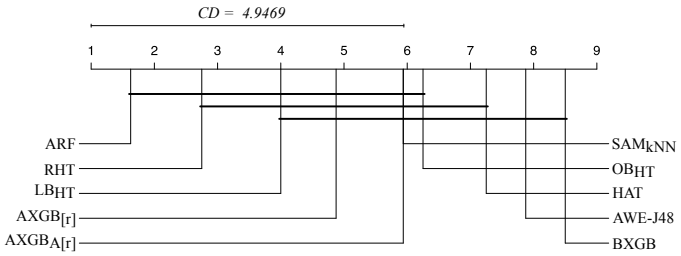


Fig. 2: Nemenyi post-hoc test (95% confidence level), identifying statistical differences between all methods in our tests.

A. Predictive Performance

We evaluate the performance of AXGB against other batch-incremental methods and against instance-incremental methods. We use prequential evaluation [22], where predictions are generated for a sample in the stream before using it to train/update the model. We use classification accuracy as the metric in our tests in order to measure performance. First, we compare the different versions of AXGB ($AXGB_{[p]}$, $AXGB_{A[p]}$, $AXGB_{[r]}$ and $AXGB_{A[r]}$) against two batch-incremental methods: BXGB and AWE-J48. The parameters used to configure these methods are available in Table II.

Results comparing against batch-incremental methods are available in Table III. We see that the overall top performer in this test is $AXGB_{[r]}$, followed by $AXGB_{A[r]}$. Next are the versions of AXGB using the push strategy. Interestingly, we find that AWE-J48 performs better than BXGB, which comes last in this test. This is noteworthy considering that the base learner in AWE-J48 (a single decision tree) is simpler than the one in BXGB (an ensemble of trees generated using XGBoost).

These tests provide insights into the different versions of AXGB. We see that, in the push-strategy versions, tracking performance to detect concept drift ($AXGB_{A[p]}$) provides a consistent advantage over the drift-unaware approach ($AXGB_{[p]}$). The reason for this is that, as expected, $AXGB_{A[p]}$ reacts faster to changes in performance: When a drift is detected, the window size is reset and new models are quickly added to the ensemble, flushing-out older models. This is not the case for methods using the replace-strategy, with $AXGB_{[r]}$ providing the best performance for most datasets. These results are significant given the compromise between the computational overhead of tracking concept drift and the gains in performance. We analyze this trade-off when discussing results of the running time tests.

TABLE III: Comparing performance of AXGB vs batch-incremental methods.

Dataset	$AXGB_{[p]}$	$AXGB_{[r]}$	$AXGB_{A[p]}$	$AXGB_{A[r]}$	BXGB	AWE-J48
AGR_a	0.919	**0.931**	0.927	0.928	0.703	0.926
AGR_g	0.896	**0.907**	0.897	0.901	0.710	0.905
AIRL	0.604	0.621	0.611	0.618	**0.641**	0.599
ELEC	0.718	0.739	0.740	**0.747**	0.702	0.614
$HYPER_f$	0.822	**0.847**	0.825	**0.847**	0.756	0.777
SEA_a	0.865	**0.875**	0.866	0.874	0.856	0.860
SEA_g	0.863	**0.873**	0.863	0.872	0.857	0.860
WEATHER	0.765	**0.774**	0.767	0.747	0.737	0.712
avg. rank	4.188	**1.438**	3.063	2.313	5.125	4.875

Next, we compare AXGB against instance-incremental methods. Results are shown in Table IV. For AXGB, we only show results of $AXGB_{[r]}$ and $AXGB_{A[r]}$. We see that the top performer in this test is ARF, closely followed by RHT. AXGB's performance is not on par with that of the top performers, but it is important to note that (i) these results are consistent with those in [14], where instance-incremental methods outperform batch-incremental methods, and (ii) both $AXGB_{[r]}$ and $AXGB_{A[r]}$ are placed in the top tier between LB_{HT} and SAM_{kNN}.

The corrected Friedman test with $\alpha = 0.05$ indicates that there are statistical significant differences between the methods in Table III and Table IV . The follow-up post-hoc Nemenyi test, Figure 2, indicates that there are no significant differences among the methods in the top tier. We believe that these findings serve to indicate the potential of eXtreme Gradient Boosting for data streams.

B. Hyper-parameter Relevance

As previously mentioned, hyper parameters play a key role in the performance of XGB. Thus, we also need to consider their impact in AXGB. In order to do so, we present results

TABLE V: Parameter grid used to evaluate hyper-parameters relevance.

Parameter	Values
max depth	1, 5, 10, 15
learning rate	0.01, 0.05, 0.1, 0.5
ensemble size	5, 10, 25, 50, 100
max window size	512, 1024, 2048, 4096, 8192
min window size	4, 8, 16

obtained by running multiple tests on different combinations of key parameters: the maximum depth of the trees, the learning rate (eta), the ensemble size, and the maximum and minimum window size. To cover a wide range of values for each parameter, we use a grid search based on the grid parameters specified in Table V. The parameter grid corresponds to a total of $4 \times 4 \times 5 \times 5 \times 3 = 1200$ combinations. For this test, we compare the following XGB-based methods: $\text{AXGB}_{[p]}$, $\text{AXGB}_{A[p]}$ and BXGB.

For establishing the effect of parameter tuning, the test is split into two phases: training and optimization on the first 30% of the stream—using this validation data to evaluate all parameter combinations in the grid and choosing the best one using prequential evaluation of classification accuracy—and performance evaluation on the remaining 70% of the stream to establish accuracy of the parameter-optimized algorithm by evaluating the algorithm with the identified parameter settings using prequential evaluation on this remaining data. The ensemble model is trained from scratch in this second phase. This strategy is limited in the sense that the nature of the validation data, including concepts drifts, is assumed to be similar to that of the remaining data. Nonetheless, it provides insights into the importance of hyper parameters.

Results from this experiment are available in Table VI. Reported results correspond to measurements obtained with parameter tuning (*Tuning*) vs. reference results (*Ref*) obtained using the fixed parameters in Table II, building an ensemble from scratch on the same 70% portion of the stream.

We can see that optimizing hyper parameters clearly benefits all methods. As expected, hyper-parameters can provide an advantage over other methods. In this case, under-performers are now on par or above LB_{HT}. Surprisingly, BXGB obtains the largest boost in performance and is now the method that performs best. When analyzing the parameter configurations (detailed results not included due to space constraints), we see that BXGB favors smaller values for max window size, learning rate and max depth. The observed increase in performance can be attributed to the impact of the hyper parameters on the base learner in BXGB (batch XGB models), remembering that BXGB is an ensemble of ensembles. Another factor to consider is the small window sizes. In practice, having smaller windows means that models are replaced faster as the stream progresses and this can ameliorate the lack of drift awareness to some degree. It is reasonable that the same applies to the reduction in the performance gap between $\text{AXGB}_{[p]}$ and $\text{AXGB}_{A[p]}$. In the case of AXGB, our results show that the learning rate has a consistent impact on performance (lower is better), followed by max window size and max depth. Finally, our tests reveal the contrast in the impact of the ensemble size on the two versions of AXGB. While $\text{AXGB}_{[p]}$ benefits from a smaller ensemble size, the opposite applies to $\text{AXGB}_{A[p]}$. This supports the intuition that drift-aware methods can benefit from larger ensembles (to build complex models) which adapt faster in the presence of drift by triggering the corresponding ensemble update mechanism. On the other hand, batch-incremental methods without explicit drift detection mechanisms rely on their natural ability to adapt, which can be counterproductive with large ensemble sizes. It is important to note that although BXGB is the top performer in this test, it is not efficient in terms of resources (time and memory), which affects stream applications where resources are limited.

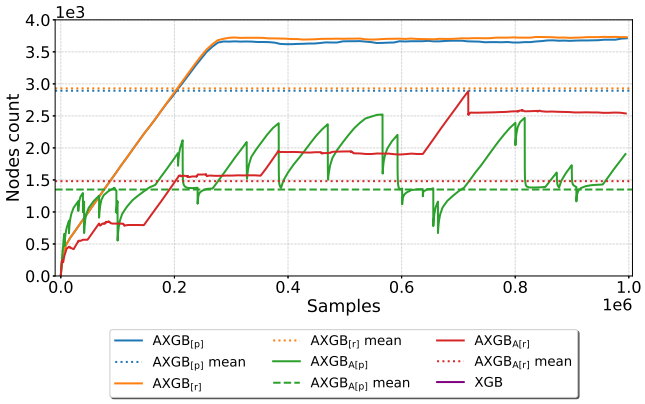
TABLE VI: Parameter tuning results.

Dataset	*Ref* $\text{AXGB}_{[p]}$	*Tuning* $\text{AXGB}_{[p]}$	*Ref* $\text{AXGB}_{A[p]}$	*Tuning* $\text{AXGB}_{A[p]}$	*Ref* BXGB	*Tuning* BXGB
AGR_a	0.881	0.927	**0.933**	0.931	0.727	0.930
AGR_g	0.898	0.906	0.902	0.905	0.728	**0.909**
AIRL	0.616	0.627	0.588	0.628	0.632	**0.639**
ELEC	0.713	0.736	0.658	0.739	0.631	**0.742**
HYPER_f	0.816	0.873	0.833	0.876	0.754	**0.904**
SEA_a	0.879	0.889	0.881	**0.892**	0.854	0.890
SEA_g	0.877	0.888	0.878	**0.889**	0.855	0.888
WEATHER	0.755	0.767	0.758	0.765	0.703	**0.782**
average	0.804	0.827	0.804	0.828	0.736	**0.835**

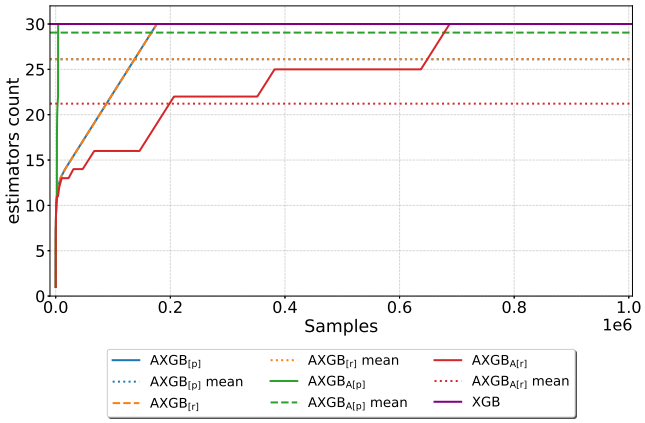
C. Memory Usage and Model Complexity

In this section, we analyze memory usage of the proposed methods during the learning process. For this purpose, we count the number of nodes in the ensemble, including both leaf nodes and internal nodes of each tree. This approach also provides some intuition regarding the model's complexity. We perform this test on a synthetic dataset with 40 features (only 30 of which are informative) and 5% noise, corresponding to the *Madelon* dataset, described in [23]. We use 1 million samples for training and calculate the number of nodes in the ensemble to get an estimate of the model size. Models are trained using the following configuration: ensemble size = 30, max window size = 10K, min window size = 1, learning rate = 0.05, max depth = 6. We measure the number of nodes as new members of the ensemble are introduced. Results from this test are available in Figure 3, and serve to compare the number of nodes in the batch model vs. the stream models. For reference, the number of nodes in the XGB model is 12.7K (outside the plot area). It is important to note that this number is constant since it represents the size of the model trained on all the data.

In Figure 3a, we see that $\text{AXGB}_{[p]}$ and $\text{AXGB}_{[r]}$ have similar behaviors. As the stream progresses, the number of nodes added to the ensemble increases until reaching a plateau. This is expected since new models are trained on larger windows of data. The plateau corresponds to the region where the ensemble is complete and old members of the ensemble are replaced by new members trained on equally large windows.



(a) Total nodes in the ensemble.



(b) Number of ensemble members.

Fig. 3: Insight into ensemble complexity by number of nodes and ensemble members over the stream. For reference, an XGB model with 30 ensemble members trained on all the data has 12.7K nodes (outside the plot area).

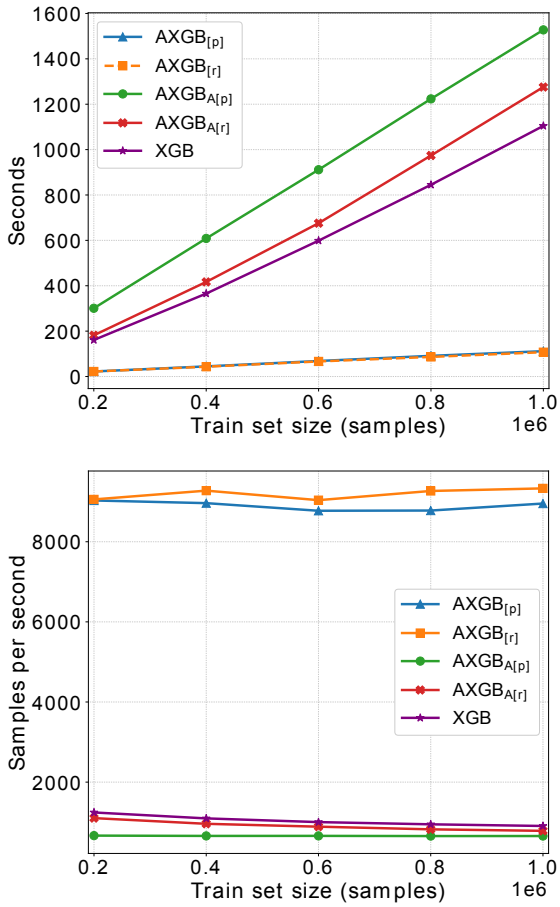


Fig. 4: Training time (top) and throughput (bottom) test results.

On the other hand, AXGB$_{A[p]}$ and AXGB$_{A[r]}$ also exhibit an incremental increase in the number nodes over the stream—at a lower rate than the AXGB versions—but with some interesting differences. In AXGB$_{A[p]}$, we see multiple drop points in the nodes count, which can be attributed to the ensemble update mechanism. When drift is detected, the window size is reset and new models are pushed into the ensemble, in other words, simpler models are quickly introduced into the ensemble. In contrast, the number of nodes in AXGB$_{A[r]}$ increases steadily. This difference in number of nodes can explain the difference in performance between AXGB$_{[r]}$ and AXGB$_{A[r]}$ discussed in Sec. V-A.

We also analyze AXGB by counting the number of models in the ensemble across the stream, shown in Figure 3b. Notice that the number of models reach the maximum value when the ensemble is full; from that point on, new models replace old ones. As anticipated, we see that AXGB$_{A[p]}$ fills the ensemble quickly at the beginning of the stream because concept drift detection triggers the reset of the window size and speeds up the introduction of new models. AXGB$_{[p]}$ and AXGB$_{[r]}$ fill the ensemble at a slower rate and finish filling the ensemble before the 200K mark. This is in line with our expectations given the introduction of new models trained on increasing window sizes as defined in Eq. 5. Finally, AXGB$_{A[r]}$ is the slowest to fill the ensemble at around the 700K mark. This is expected given that upon drift detection, AXGB$_{A[r]}$ starts replacing the oldest models of the ensemble.

It is important to mention that additional memory resources are used by the different AXGB variants given their batch-incremental nature: mini-batches are accumulated in memory before they are used to fit a tree. In this sense, other things being equal, instance-incremental methods are more memory efficient. However, our results show that all versions of AXGB keep the size of the model under control, a critical feature when facing theoretically infinite data streams.

D. Training Time

Finally, we measure training time for the different versions of AXGB. We use as reference the time required to train an XGB model on the *Madelon* dataset used in the model complexity test. Models are trained using the following configuration: ensemble size = 30, max window size = 10K, min window size = 1, learning rate = 0.05, max depth = 6. We used the following dataset sizes: 200K, 400K, 600K,

800K and 1M. Results correspond to the average time after running the experiments 10 times for each dataset size and for each classifier. Measurements are shown in Figure 4 in terms of time (seconds) and in Figure 4 in terms of throughput (samples per second). These tests show that the fastest learners are $\text{AXGB}_{[p]}$ and $\text{AXGB}_{[r]}$, both showing small change in training time as the number of instances increases. This is an important feature given that training time plays a key role in stream learning applications. On the other hand, $\text{AXGB}_{A[p]}$ and $\text{AXGB}_{A[r]}$ have similar behaviour in terms of training time compared to XGB while being slightly slower. This can be attributed to the overhead from the drift-detection process, which implies getting predictions for each instance and keeping the drift detector statistics. Additionally, we see that $\text{AXGB}_{A[p]}$ is the slowest classifier, which might be related to the overhead incurred by predicting using more ensemble members, given that the ensemble is quickly filled as previously discussed.

VI. Conclusions

In this paper, we propose an adaptation of the eXtreme Gradient Boosting algorithm (XGB) to evolving data streams. The core idea of Adaptive XGBoost (AXGB) is the incremental creation/update of the ensemble, i.e., weak learners are trained on mini-batches of data and then added to the ensemble. We study variations of the proposed method by considering two main factors: concept drift awareness and the strategy to update the ensemble. We test AXGB against instance-incremental and batch-incremental methods on synthetic and real-world data. Additionally, we consider a simple batch-incremental approach (BXGB) consisting of ensemble members that are full XGB models trained on consecutive mini-batches. From our tests, we conclude that $\text{AXGB}_{[r]}$ (the version that performs model replacement in the ensemble but does not include explicit concept drift awareness) represents the best compromise in terms of performance, training time and memory usage.

Another noteworthy finding from our experiments is the good predictive performance of BXGB after parameter tuning. If resource consumption is a secondary consideration, this approach may be a worthwhile candidate for application in practical data stream mining scenarios, particularly considering that our parameter tuning experiments did not investigate optimizing the size of the boosted ensemble for each mini-batch in BXGB. (The size of each sub-ensemble was fixed at 30 members.) Overall, despite the limitations of mini-batch-based data stream mining, and its drawbacks compared to instance-incremental methods, it appears that XGB-based techniques are promising candidates for data stream applications. In a similar way, we believe AXGB is an interesting alternative to XGB for some applications given its efficient management of resources and adaptability.

References

[1] W. N. Street and Y. Kim, "A streaming ensemble algorithm (sea) for large-scale classification," in *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2001, pp. 377–382.

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

[3] N. Oza, "Online Bagging and Boosting," in *IEEE International Conference on Systems, Man and Cybernetics*, vol. 3. IEEE, 2005, pp. 2340–2345.

[4] 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*, 2010, pp. 135–150.

[5] A. Bifet and R. Gavaldà, "Learning from Time-Changing Data with Adaptive Windowing," *Proceedings of the 2007 SIAM International Conference on Data Mining*, pp. 443–448, 2007.

[6] A. Bifet, E. Frank, G. Holmes, and B. Pfahringer, "Ensembles of Restricted Hoeffding Trees," *ACM Transactions on Intelligent Systems and Technology*, vol. 3, no. 2, pp. 1–20, feb 2012.

[7] V. Losing, B. Hammer, and H. Wersing, "KNN classifier with self adjusting memory for heterogeneous concept drift," *Proceedings - IEEE International Conference on Data Mining, ICDM*, pp. 291–300, 2017.

[8] 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. 9-10, pp. 1469–1495, 2017.

[9] L. Breiman, D. Wolpert, P. Chan, and S. Stolfo, "Pasting Small Votes for Classification in Large Databases and On-Line," *Machine Learning*, vol. 36, pp. 85–103, 1999.

[10] E. Frank, G. Holmes, and R. Kirkby, "Racing committees for large datasets," *Discovery Science*, pp. 153–164, 2002.

[11] R. Polikar, L. Udpa, S. S. Udpa, and V. Honavar, "Learn++: An incremental learning algorithm for supervised neural networks," *IEEE Transactions on Systems, Man and Cybernetics Part C: Applications and Reviews*, vol. 31, no. 4, pp. 497–508, 2001.

[12] Y. Freund and R. E. Schapire, "A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting," *Journal of Computer and System Sciences*, pp. 119–139, 1997.

[13] H. Wang, W. Fan, P. S. Yu, and J. Han, "Mining concept-drifting data streams using ensemble classifiers," in *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '03*, vol. 42. New York, New York, USA: ACM Press, 2003, p. 226.

[14] J. Read, A. Bifet, B. Pfahringer, and G. Holmes, "Batch-incremental versus instance-incremental learning in dynamic and evolving data," *Lecture Notes in Computer Science*, vol. 7619, pp. 313–323, 2012.

[15] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16. New York, NY, USA: ACM, 2016, pp. 785–794.

[16] J. Demšar, "Statistical comparisons of classifiers over multiple data sets," *J. Mach. Learn. Res.*, vol. 7, p. 1–30, Dec. 2006.

[17] S. García and F. Herrera, "An extension on "statistical comparisons of classifiers over multiple data sets" for all pairwise comparisons," *Journal of Machine Learning Research*, vol. 9, pp. 2677–2694, 2009.

[18] J. Montiel, J. Read, A. Bifet, and T. Abdessalem, "Scikit-Multiflow: A Multi-output Streaming Framework," *Journal of Machine Learning Research*, vol. 19, pp. 1–5, 10 2018.

[19] A. Bifet, G. Holmes, R. Kirkby, and B. Pfahringer, "Moa: Massive online analysis," *Journal of Machine Learning Research*, vol. 11, no. May, pp. 1601–1604, 2010.

[20] C. C. Aggarwal, S. Y. Philip, J. Han, and J. Wang, "-a framework for clustering evolving data streams," in *Proceedings 2003 VLDB Conference*. Elsevier, 2003, pp. 81–92.

[21] G. Hulten, L. Spencer, and P. Domingos, "Mining time-changing data streams," in *ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2001, pp. 97–106.

[22] A. P. Dawid, "Present position and potential developments: Some personal views: Statistical theory: The prequential approach," *Journal of the Royal Statistical Society. Series A (General)*, pp. 278–292, 1984.

[23] I. Guyon, J. Li, T. Mader, P. A. Pletscher, G. Schneider, and M. Uhr, "Competitive baseline methods set new standards for the NIPS 2003 feature selection benchmark," *Pattern Recognition Letters*, vol. 28, no. 12, pp. 1438–1444, 2007.