STREAMRHF: Tree-Based Unsupervised Anomaly Detection for Data Streams

Stefan Nesic
Télécom Paris
Paris, France
stefan.nesic@telecom-paris.fr

Andrian Putina
Télécom Paris
Paris, France
andrian.putina@telecom-paris.fr

Maroua Bahri
Inria Paris
Paris, France
maroua.bahri@inria.fr

Alexis Huet
Huawei Technologies Co., Ltd.
Paris, France
alexis.huet@huawei.com

Jose Manuel Navarro
Huawei Technologies Co., Ltd.
Paris, France
jose.manuel.navarro@huawei.com

Dario Rossi
Huawei Technologies Co., Ltd.
Paris, France
dario.rossi@huawei.com

Mauro Sozio
Télécom Paris
Paris, France
mauro.sozio@telecom-paris.fr

***Abstract*—We present STREAMRHF, an unsupervised anomaly detection algorithm for data streams. Our algorithm builds on some of the ideas of Random Histogram Forest (RHF) [1], a state-of-the-art algorithm for batch unsupervised anomaly detection. STREAMRHF constructs a forest of decision trees, where feature splits are determined according to the kurtosis score of every feature. It irrevocably assigns an anomaly score to data points, as soon as they arrive, by means of an incremental computation of its random trees and the kurtosis scores of the features. This allows efficient online scoring and concept drift detection altogether. Our approach is tree-based which boasts several appealing properties, such as explainability of the results [2].**

We conduct an extensive experimental evaluation on multiple datasets from different real-world applications. Our evaluation shows that our streaming algorithm achieves comparable average precision to RHF while outperforming state-of-the-art streaming approaches for unsupervised anomaly detection with furthermore limited computational complexity.

***Index Terms*—Data streams, Unsupervised learning, Anomaly detection, Random histogram**

I. Introduction

Several emerging applications and devices produce and accumulate a massive volume of data at an ever increasing rate in the form of streams. As a result, data stream mining has become common in different domains, such as network security, traffic management, health monitoring, and social networks. Streaming algorithms must satisfy strict memory requirements (typically sublinear in the size of the input dataset) and have fast processing times. This is in contrast with the classical static – or batch – setting where data can be randomly accessed multiple times with less strict processing time or memory constraints. Anomaly detection is a widely studied problem in machine learning and data mining. Roughly speaking, it consists of identifying data points that "significantly" deviate from the other points in a given dataset. Historically, outlier detection served mainly as a tool to filter out noise, which could impair the training of a machine learning algorithm. Nowadays, it plays a more noble role, as the machine learning community realized that anomalies are often associated with interesting or rare events.

Applications of anomaly detection are vast, encompassing network security and maintenance, data storage, finance, and medicine. For example, anomaly detection algorithms provide important primitives in the detection of intrusions in networks, frauds in financial transactions, data leaks, as well as life-threatening situations in patient monitoring systems. Often data arrives in a continuous fashion, in real-time, and in potentially infinite supply [3]. In many real-world applications, such as the detection of failures or attacks in networks, we seek to detect the issue as soon as possible in order to minimize the damage. As a result, anomaly detection algorithms for data streams must boast fast processing times as well as meeting strict storage requirements, while delivering satisfactory results. A large number of approaches have been developed over the years. Supervised anomaly detection can be used when large amounts of labeled data are readily available. However, due to the difficulty in obtaining labeled data, unsupervised approaches preserve their appeal.

Despite the efforts made in recent years, most anomaly detection algorithms are designed in the classical offline setting. Among the most successful algorithms we mention Local Outlier Factor (LOF) [4], Isolation Forest (iForest) [5], One-Class Support Vector Machines (OCSVM or 1-SVM) [6], and more recently, Random Histogram Forest (RHF) [1]. In particular, the latter methods have emerged as state-of-the-art approaches for unsupervised anomaly detection in the static setting. However, when applied on data streams, static anomaly detection algorithms fail to process potentially infinite sequences of instances because of their evolving nature [7]. To cope with this issue and address the streaming framework requirements, several models of computation have emerged [8]. This includes *single-pass* processing where instances should be processed only once (or a couple of times), the use of a *sliding window* [9] of a fixed size, where only the most recent instances from the stream are retained, and *dimensionality reduction* [10] to reduce the number of dataset features. XSTREAM [11] and Robust Random Cut Forest (RRCF) [12] have emerged as state-of-the-art algorithms for evolving data streams.

In our work, we present STREAMRHF, an unsupervised anomaly detection algorithm for data streams, which is built on some of the ideas of the RHF method. Contrary to RHF, STREAMRHF assigns to each data point an anomaly score *online* as soon as it arrives. The online setting also imposes that such a score cannot be changed afterwards as new data points arrive. The scoring is done by building a decision tree where feature splits are determined according to the kurtosis score (aka fourth moment) of every feature. As new data points arrive, the random trees and the kurtosis scores are updated incrementally, which allows for an efficient computation of the anomaly scores for the remaining data in the stream. An additional sliding window allows to retain only the most recent data points in the stream. Another appealing property of our approach is that it is tree-based (contrary to XSTREAM) which allows for explainability [2].

We conduct an extensive experimental evaluation on 18 datasets containing up to 600k instances from a diverse set of real-world problems. The source code used in our analysis can be found at the following address: `https://github.com/stefannesic/streamRHF`. Our evaluation shows that our streaming algorithms achieve comparable average precision to RHF, while outperforming state-of-the-art stream approaches for unsupervised anomaly detection, such as XSTREAM and RRCF. Moreover, STREAMRHF does not need manual tuning of any critical parameter, hence it can be directly applied on a wide range of datasets without manual drill-down. This appealing property along with its efficiency makes it particularly suitable in a deployment context.

The rest of the paper is organized as follows. In Section II we discuss the related work for anomaly detection. In Section III, we present our main algorithm, while in Section IV we show our experimental evaluation performed on a large set of datasets. Finally, we recap our conclusions in Section V.

II. Related Work

The literature on anomaly detection is vast. In this work, we mainly focus on unsupervised anomaly detection approaches, since in many scientific disciplines, especially intrusion detection and fraud detection, ground truth data is generally missing [13]. In this context, different algorithms have been proposed for unsupervised anomaly detection for data streams but only few of them outperform multiple standard anomaly detection algorithms and hence, are considered state-of-the-art, such as HST [7], RRCF [14], and XSTREAM [11].

Half-Space-Trees (HST) [7] is an ensemble of trees with a fixed depth (height) that computes the anomaly scores based on the sample counts and densities of the nodes. HST uses two windows of equal size (256), where the trees trained on the previous window are used to observe instances in a given batch. Simultaneously, new trees are trained on the current window and once all instances from the current window are processed, new trees replace the old ones. The latter strategy is employed to handle concept drifts[1] in data streams. HST

[1]Changes in the data distribution, with the current concept no longer being representative for the next incoming data.

TABLE I
Summary of streaming algorithms for anomaly detection. Capabilities of the algorithms are expressed in terms of their ability to immediately update the scoring for each incoming data point (point-by-point reactivity), to manage concept drift by forgetting older information, and to keep a white-box approach (tree-based).

Algorithm	**STREAMRHF (this work)**	**XSTREAM**	**RRCF**	**HST**
Tree-based	✓	✗	✓	✓
Point-by-point reactivity	✓	✗	✓	✗
Concept drift	✓	✓	✗	✓

assumes that the data is scaled, such that features values are bounded in [0,1]. The latter is a more relaxed condition, especially in the presence of concept drifts [15].

Robust Random Cut Forest (RRCF) [12] is a stream detector based on iForest [5] and used on dynamic data streams by treating different dimensions independently. RRCF preserves pairwise distance which is important for computation and likewise for anomaly detection. Unfortunately, RRCF suffers from scalability issues, as shown by our experimental evaluation (provided in Section IV).

Lightweight online detector of anomalies (Loda) streaming method [15] consists of an ensemble of h one-dimensional histograms, where each histogram with sparse projections in Loda provides an anomaly score on a randomly generated subspace. Projection vectors diversify individual histograms, which is a necessary condition to enhance the performance of individual classifiers in high-dimensional spaces. Thanks to the use of random projection that simplifies the complexity of all operations, this method has been shown to be $7-8$ times faster than HST. Due to the fixed number of bins for the one-dimensional histograms, Loda cannot however handle growing feature space.

In [11], authors proposed XSTREAM, a window-based detection method particularly effective in feature-evolving data streams[2] through the use of a stream random projection scheme to handle high-dimensionality. The projection is performed by recursively constructing partitions with splits into small flexible bins, unlike Loda that cannot handle growing feature space due to the fixed number of bins for the one-dimensional histograms. XSTREAM is able to detect anomalies accurately and handle streams of variable dimensionality and missing values.

Recently, an ensemble method has been developed, RHF [1], which is based on a forest of trees. RHF emerged to be a state-of-the-art approach for unsupervised anomaly detection in the static batch setting [1]. To the best of our knowledge, no streaming version of this novel algorithm has been developed, which is one of the purposes of our work. Compared to existing algorithms, extending RHF with streaming capabilities confers several advantages. First of all, the core of the algorithm remains based on trees that are in-

[2]Streams with newly-emerging features and changing feature values.

dividually white-box models and explainable by nature. Then, the construction of RHF using kurtosis enables the possibility to update the trees incrementally in a continuous way (without the need to rebuild the whole forest) and to achieve point-by-point reactivity. Finally, STREAMRHF introduces a sliding window to handle concept drift and to bound the memory requirements.

A summary of the main capabilities of STREAMRHF compared to the other streaming approaches is available in Table I. Most of the previous work considers alternating windows, such as XSTREAM and HST, that prevent an immediate update of the model, while other works, such as RRCF, are not able to react to concept drift. STREAMRHF has both point-to-point reactivity and concept drift management.

The aforementioned algorithms, notably XSTREAM, RRCF, and HST, serve the purpose of common baselines for comparison since they are the most recent and effective state-of-the-art stream anomaly detection methods.

III. Unsupervised Random Histogram Forest for Data Streams

Data streams are defined as continuous sequences of incoming instances, where each instance is composed of multiple attributes taking numerical or binary values. In this paper, we develop an unsupervised anomaly detection algorithm that aims to detect anomalies in data streams. Our approach is built on some of the ideas of RHF. In the following section, we recall how RHF works and then, we discuss how to address the challenges in developing a streaming algorithm for anomaly detection.

A. Random Histogram Forest

RHF is an ensemble of t trees of maximum height h that partitions data randomly with a higher probability of partitioning on features (aka attributes) whose distribution has a high value of kurtosis, the fourth standardized moment in statistics. The intuition behind the partitioning is that instances that end up in groups with fewer instances are more likely to be anomalies. For each split, an attribute, a_s, and a value, val_a, are selected from the range of possible values of a_s. Then, data with a value for the attribute a_s less than val_a becomes the left child and the remainder, the right child.

For a dataset $(X_a)_{a\in[\![0,d]\!]}$ consisting of $d+1$ attributes, the tree split is defined as follows [1]:

1) The sum of the kurtosis K of all features $(X_a)_{a\in[\![0,d]\!]}$ is computed:

$$K_s = \sum_{a=0}^{d} \log[K(X_a)+1]. \tag{1}$$

2) A random number r is chosen

$$r \sim \text{Uniform}([0, K_s)). \tag{2}$$

3) The split attribute is selected based on this r value,

$$a_s = \arg\min \left(k \in [\![0,d]\!] \,\middle|\, \sum_{a=0}^{k} \log[K(X_a)+1] > r \right), \tag{3}$$

while the value val_a is taken at random (uniformly) in the observed range of that feature.

The trees are split until either the maximum height is reached or the kurtosis of each attribute is zero, meaning that all the remaining instances are duplicates. Every instance, i, is then scored on each tree, T, in the forest based on the number of instances in their leaf, S_l, or more precisely, the *Information Content*:

$$w_i^T = \log\left(\frac{1}{|S_l|}\right). \tag{4}$$

The score of i, w_i, is then calculated as the sum of the t tree scores:

$$w_i = \sum_{T} w_i^T. \tag{5}$$

B. Point-By-Point Reactivity

When adapting an algorithm to the stream setting, the well-known issue of concept drift must be addressed as the data distribution can change over time and the initial model may no longer accurately reflect the distribution of data. Thus, we need to efficiently update our model so as to take into account the current distribution of data. One naive solution would be to recompute from scratch every tree in the forest as soon as a new data point arrives. However, this is not a viable solution as it is clearly very expensive. Alternatively, we could rebuild a subtree as soon as the current kurtosis value of the corresponding attribute "significantly" deviates from its value computed during the last rebuild. Such a solution would require to introduce a threshold on the difference between the current kurtosis value and the one computed in the last rebuild. Unfortunately, it is unclear how to set such a threshold *a priori*, in that, it might depend on the input data.

Therefore, we strive to develop a parameter-free algorithm which can efficiently build a random histogram forest as close as possible to the one that the batch algorithm would compute at every single update operation. The initial model is built on the first n initial points of the data stream using the same RHF algorithm and scored using a function that is detailed in Equations 4 and 5. This model will serve as the base to our proposed algorithm described later in this section. Next, when an instance x arrives, we proceed as follows. Starting from the root, we first check if by inserting x, the splitting attribute of the current node will change. If this is the case, we rebuild the current subtree. Otherwise, we proceed recursively in either the left subtree or the right subtree, if the value of the splitting attribute node in x is at most its splitting value or larger, respectively. This process is iterated recursively until a leaf is reached, in which case, the score of x is calculated using Equation 4. Consequently, this allows us to rebuild only a partial subtree, as opposed to a naive algorithm that rebuilds each time the whole tree.

Algorithm 1 Insert(node, i, h, z)

```
Input: node: node of random histogram tree, i: instance, h: max tree height, z: the seed of array of $2^h$
Output: Random Histogram Tree rht
1:  if node is not Leaf then
2:      kvalues = kurtosis($node.data \cup i$);  ▷ same function as in the batch, but the seed of every node is initialized using values of z
3:      $a_{split}$ = get-attribute(kvalues, z[node.id]);
4:      if $node.attribute \neq a_{split}$ then
5:          return RHT-build($node.data \cup i$, node.height)
6:      if $i[node.attribute] \leq node.value$ then
7:          node.left = Insert(node.left, i, h, z)
8:      else
9:          node.right = Insert(node.right, i, h, z)
10: else
11:     if $node.height = h$ then
12:         node.data = $node.data \cup i$
13:         return node
14:     else
15:         return RHT-build( $node.data \cup i$, node.height)
```

In fact, the splitting attribute for a given node is determined by a random value r, called a *seed*. To efficiently determine when the current subtree needs to be rebuilt, we initialize randomly the value r for every node in every tree and then maintain its value throughout the entire data stream. Then, for every node, the value r will guide the choice of the splitting attribute. Let $a_{s_{\text{current}}}$ be the existing split attribute in the model. We therefore select the splitting attribute as follows:

$$a_{s_{\text{new}}} = \arg\min \left(k \left| \sum_{a=0}^{k} \log[K(X'_a) + 1] > r \right. \right), \quad (6)$$

where $X' = X \cup x$. If $a_{s_{\text{current}}} \neq a_{s_{\text{new}}}$, the sub-tree is rebuilt from scratch using the batch algorithm.

STREAMRHF is consequently able to adapt to changes in the distribution of the data upon insertion of new instances which provides an advantage over the naive approach. The details for the insertion of the new instances are provided in Algorithm 1.

C. Concept Drift Management

Since the random seeds are fixed in the beginning, the model constructed at each insertion is the same as the original batch algorithm without the need to build a forest from scratch at each insertion. The aim is to allow the model to forget about the old inserted instances, which also accounts for possible memory constraints. For this purpose, our algorithm employs two windows: a reference window and a current window. The reference window is of a fixed size and is filled with the first n instances from the stream. The current window is initially empty and is gradually filled by each of the upcoming instances, x_i, that are inserted into the model using Algorithm 1. Once the current window is filled with n instances, the reference window is overwritten by the observations in the current window which is in turn emptied. The model is rebuilt from scratch with the new reference window and the process is repeated at each time the current window is filled. Such strategy turns out to be effective when dealing with concept drifts, because we are constantly keeping the most recent instances which are the most suitable ones for representing the current concept in the data stream. In Algorithm 2, we present the pseudocode of STREAMRHF.

Algorithm 2 STREAMRHF(X, h, t, n)

```
Input: X: stream, h: height, t: number of trees, n: window size
Output: scores, one for each instance in X
1:  scores = [];
2:  for i in 0..t do
3:      for i in 0..$2^h - 1$ do
4:          z[i,j]= RandomSeed();  ▷ set seeds, one per node per tree, assuming complete tree
5:  forest = RHF-build(X[0:n], h, t, z);
6:  scores[0:n] = RHF-score(forest, X[0:n]);
7:  for i in X[n+1:end] do
8:      for all tree in forest do
9:          tree = Insert(tree, i, z[tree.id]);
10:     scores[i] = RHF-score(forest, i)
11:     if i % n == 0 then  ▷ reference window
12:         forest = RHF-build(X[i-n:i], h, t);
    return scores
```

D. Running Time & Space Requirements

The worst-case running time and the space requirements of STREAMRHF can be evaluated as follows. Let w be the size of the sliding window, d the number of features, h the maximum height of every tree, and t be the total number of trees in our forest. STREAMRHF requires $O(wdt)$ space, in that, at each point in time, for every tree, we only need to store all points in the sliding window. The running time of the initialization step is in $O(wdth)$, as in the worst-case we might need to process all the points in the sliding window for every level of a given tree. The number of operations per update can be as large as the initialization step, however, this happens only if we rebuild the whole tree from the root, which does not happen often according to our experimental evaluation. Indeed, in most of the cases we rebuild only relatively small subtrees with height $\bar{h} < h$, which gives a running time of $O(wdt\bar{h})$ for such an update. For future work, it would be interesting to provide a tighter analysis of the running time of our algorithm on the average case.

IV. Experimental Evaluation

We conduct an extensive experimental evaluation on 18 different datasets containing more than 600k instances from a diverse set of real-world applications. We evaluate our approach against XSTREAM, RRCF, and HST, which emerged as state-of-the-art approaches for our problem.

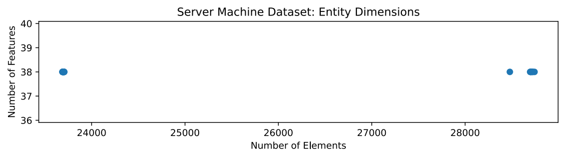

Fig. 1. The SMD multivariate time series dataset is divided into 28 entities with 38 features but varying element counts (708,420 total elements).

A. Experiment Settings

Environment. The experiments were conducted on a Ubuntu 20.04.2 LTS server equipped with 144 CPUs of model Intel(R) Xeon(R) Gold 6154 @ 3.00GHz and 264 GB of RAM. The implementation of the STREAMRHF algorithm was done in Python 3 and Cython. The implementations of RHF and XSTREAM were taken from the GitHub repositories of the respective authors. The C++ version from the repository of XSTREAM[3] is the one adapted to streaming data. The Python implementations of Robust Random Cut Forest[4] and HST[5] were used. In order to improve the reproducibility of our proposed STREAMRHF algorithm, the source code used in our analysis is also made available[6].

Parameters. All parameters of RHF and XSTREAM were set in accordance with the instructions of the respective authors in their original work. In particular, the parameters of STREAMRHF and HST were set with the same values, $T = 100$ and $H = 5$. For XSTREAM, 100 half-space chains were used with a projection size of 100 and a depth of 15. Although the paper uses 1000 chains as well, this setting is much more costly in terms of execution time. The RRCF parameters were set to 100 trees and shingles were not used. The initial sample size is always equal to the window size. Window sizes were defined as percentages of the total number of instances in the dataset, e.g., 1% or 5%. The experiments were run on 10 iterations and the mean is reported.

Datasets. To assess the effectiveness of our proposed algorithm on a wide range of real-world applications, we conduct an extensive experimental evaluation over diverse datasets. In particular, we consider 17 publicly available benchmark non-temporal datasets containing between 1920 and 623,091 instances with 3 to 166 features. Non-temporal datasets were separated into two groups, "Small", datasets with less than 20k instances, and "Large", the remainder. This separation occurs due to the computational cost of the latter group. The datasets used are described in Table II. The *http_logged*, *kdd_ftp*, *kdd_http*, *kdd_http_dct*, and *kdd_smtp_all* datasets were constructed from the KDD'99 Cup dataset [16] using the same procedure as in [1]. The remaining datasets are from the UCI [17] or ODDS repositories [18]. Non-temporal datasets were shuffled at each iteration in order to reduce bias in datasets for one algorithm or another. In addition, we evaluated the algorithms on a publicly available benchmark multivariate time series dataset from KDD'19, *Server Machine Dataset (SMD)* [19], that is composed of 28 entities totaling 708,420 elements and 38 features. Only the labeled test set data was used from this dataset. As shown in Figure 1, SMD is in fact composed of 28 separated entities or subdatasets with varying element counts between roughly 23,687 to 28,726 instances each. Algorithms were evaluated on each individual entity of SMD, as recommended by the authors of the dataset.

TABLE II
OVERVIEW OF THE DATASETS.

Type	Dataset	Samples	Features	Anomalies	Duplicates
Small[1]	aabalone	1920	7	1.5%	0
	annthyroid	7200	6	2.3%	0
	kdd_ftp	5214	3	26.7%	79.7%
	magicgamma	19020	10	35.1%	1.7%
	mammography	11183	6	2.3%	2.3%
	mnist	7603	100	9.2%	0
	musk	3062	166	3.1%	0
	satellite	5100	36	1.4%	0
	satimages	5803	36	1.2%	0.03%
	spambase	4601	57	39.4%	7.4%
	thyroid	3772	6	2.4%	0
Large[2]	http_logged	567498	3	0.4%	97%
	kdd_http_dct	222027	3	0.03%	0
	kdd_http	623091	3	0.64%	98.1%
	mulcross	262144	4	10%	0
	shuttle_odds	49097	9	7%	0
	smtp_all	96554	3	0.03%	0
Time[3]	smd	708420	38	4.16%	0

[1]*datasets comprising less than 20k elements*
[2]*datasets comprising over 20k elements*
[3]*datasets where each element is a sample of a time-series*

Metrics. In [20], it was shown that, for highly-skewed datasets, the Precision-Recall (PR) curve gives more information on the accuracy than the Receiver Operator Characteristic (ROC) curve. In addition, it was shown that "a curve dominates in ROC space if and only if it dominates in PR space". Therefore, the accuracy of the various algorithms is summarized from their PR curves by the Average Precision (AP). The chosen implementation is not interpolated and is calculated as the weighted mean of precisions at each threshold, P_n, with the weight being the difference in recall at the current threshold, R_n and the previous one, R_{n-1}.

$$AP = \sum_n (R_n - R_{n-1}) P_n, \tag{7}$$

where

$$P_n = \frac{TP}{TP + FP} \text{ and } R_n = \frac{TP}{TP + FN} \tag{8}$$

are the equations for the precision and recall at the n^{th} threshold, respectively. The average insertion time is calculated as the execution time divided by the number of insertions.

[3]https://github.com/cmuxstream/cmuxstream-core/tree/master/cpp
[4]https://github.com/kLabUM/rrcf
[5]https://github.com/online-ml/river/blob/main/river/anomaly/hst.py
[6]https://github.com/stefannesic/streamRHF

TABLE III
AVERAGE PRECISION FOR BATCH AND STREAMING ALGORITHMS, ON BOTH SMALL AND LARGE DATASETS, FOR A WINDOW SIZE OF 1%.

Type	**Dataset**	**RHF**	STREAMRHF	XSTREAM	**RRCF**	**HST**
Small	abalone	$\mathbf{0.352 \pm 0.026}$	0.178 ± 0.043	0.142 ± 0.019	0.157 ± 0.029	0.29 ± 0.061
	annthyroid	0.307 ± 0.019	$\mathbf{0.425 \pm 0.012}$	0.235 ± 0.034	0.218 ± 0.008	0.109 ± 0.004
	magicgamma	$\mathbf{0.622 \pm 0.007}$	$\mathbf{0.629 \pm 0.006}$	0.629 ± 0.007	0.61 ± 0.012	0.355 ± 0.003
	mammography	$\mathbf{0.156 \pm 0.017}$	$\mathbf{0.189 \pm 0.019}$	0.201 ± 0.03	0.138 ± 0.012	0.047 ± 0.003
	mnist	$\mathbf{0.32 \pm 0.019}$	$\mathbf{0.312 \pm 0.014}$	0.292 ± 0.035	0.317 ± 0.008	0.264 ± 0.027
	musk	$\mathbf{0.987 \pm 0.023}$	0.78 ± 0.045	0.524 ± 0.09	0.776 ± 0.017	0.091 ± 0.029
	satellite	$\mathbf{0.636 \pm 0.015}$	0.61 ± 0.01	0.635 ± 0.022	0.56 ± 0.045	0.015 ± 0.0
	satimages	$\mathbf{0.928 \pm 0.01}$	0.901 ± 0.012	0.94 ± 0.008	0.777 ± 0.072	0.012 ± 0.0
	spambase	0.415 ± 0.015	$\mathbf{0.474 \pm 0.013}$	0.522 ± 0.024	0.535 ± 0.006	0.47 ± 0.015
	thyroid	$\mathbf{0.548 \pm 0.021}$	$\mathbf{0.583 \pm 0.04}$	0.235 ± 0.039	0.298 ± 0.019	0.157 ± 0.007
Large	http_logged	$\mathbf{0.969 \pm 0.002}$	0.455 ± 0.025	0.603 ± 0.053	0.313 ± 0.011	0.004 ± 0.0
	kdd_ftp	$\mathbf{0.449 \pm 0.036}$	0.279 ± 0.004	0.351 ± 0.017	0.219 ± 0.011	0.866 ± 0.002
	kdd_http_dct	$\mathbf{0.774 \pm 0.013}$	0.374 ± 0.084	0.208 ± 0.048	0.423 ± 0.025	0.035 ± 0.006
	kdd_http	$\mathbf{0.55 \pm 0.006}$	0.267 ± 0.009	0.276 ± 0.051	0.182 ± 0.014	0.021 ± 0.0
	mulcross	$\mathbf{0.729 \pm 0.033}$	$\mathbf{0.69 \pm 0.022}$	0.506 ± 0.029	0.153 ± 0.01	0.458 ± 0.027
	shuttle_odds	$\mathbf{0.935 \pm 0.005}$	0.868 ± 0.006	0.828 ± 0.022	0.492 ± 0.029	0.368 ± 0.027
	smtp_all	$\mathbf{0.944 \pm 0.023}$	$\mathbf{0.953 \pm 0.008}$	0.254 ± 0.111	0.129 ± 0.01	0.971 ± 0.019
All	mean	$\mathbf{0.625 \pm 0.039}$	0.527 ± 0.037	0.434 ± 0.036	0.37 ± 0.032	0.267 ± 0.043
	median	$\mathbf{0.612}$	0.482	0.371	0.31	0.153

Values are the average and 0.95 confidence interval over 10 runs. RHF is greyed out as it as a batch algorithm with batch scoring. The best results of streaming algorithms are in blue and best results of RHF algorithms family (batch/stream) are in **bold**. Multiple results are considered as the best ones if confidence intervals overlap.

B. Results

We start by comparing our streaming algorithm against the RHF batch algorithm in terms of precision. Figure 2 shows the mean AP, median AP, upper quartile AP, and lower quartile AP of the algorithms while using a window size of 1% (top figure) and 5% (bottom figure) of the entire dataset size, for each dataset in the "small" group. We observe that the performance of our streaming algorithm, STREAMRHF, is comparable to the batch RHF algorithm performance for the two configurations of the window size. Recall, that STREAMRHF, like other streaming algorithms, computes an anomaly score online, that is as soon as an instance arrives, without having any knowledge of the remaining data observations. Additionally, it only keeps in memory the instances that are in its window. Therefore, it is an appealing property that our streaming algorithm does not lose much in terms of average precision when compared to its batch counterpart.

Table III presents the statistical results of the algorithms for a window size of 1% on 17 non-temporal datasets. We notice that, on the overall average, the batch RHF obtains the most accurate performance for the aforementioned reasons. Nevertheless, RHF is defeated by our proposed streaming version, STREAMRHF, on 8 datasets (the values in bold).

Next, we evaluate our approach against state-of-the-art approaches for unsupervised anomaly detection in streaming on all datasets including some that contain more than 600k instances. Table III also shows that STREAMRHF has the best average precision over all the datasets in comparison with the stream anomaly detection algorithms. In particular, observe that the median AP of STREAMRHF is 30% higher than the second-best approach. This is thanks to its windowed strategy that handles evolving data while implicitly dealing with concept drifts. Figure 2 confirms that STREAMRHF delivers consistently the best results across different window sizes with XSTREAM boasting the second-best results.

TABLE IV
AP FOR VARYING WINDOW SIZES ON THE SMD TIME SERIES DATASET.

Window	STREAMRHF	XSTREAM	**RRCF**	**HST**
256	0.121 ± 0.006	0.145 ± 0.011	0.051 ± 0.004	$\mathbf{0.255 \pm 0.021}$
512	0.167 ± 0.007	0.207 ± 0.016	0.048 ± 0.004	$\mathbf{0.313 \pm 0.023}$
1024	0.221 ± 0.011	$\mathbf{0.276 \pm 0.021}$	0.046 ± 0.004	$\mathbf{0.319 \pm 0.025}$
2048	$\mathbf{0.271 \pm 0.015}$	$\mathbf{0.29 \pm 0.025}$	0.047 ± 0.005	$\mathbf{0.302 \pm 0.024}$
4096	$\mathbf{0.317 \pm 0.019}$	0.277 ± 0.022	0.047 ± 0.005	0.296 ± 0.025
20%	$\mathbf{0.327 \pm 0.019}$	0.261 ± 0.023	0.048 ± 0.005	0.296 ± 0.025
25%	$\mathbf{0.334 \pm 0.018}$	0.251 ± 0.023	0.048 ± 0.005	0.298 ± 0.025
30%	$\mathbf{0.355 \pm 0.02}$	0.242 ± 0.026	0.051 ± 0.006	0.293 ± 0.024

Values are average and 0.95 confidence interval over 10 runs, best AP in **bold**. Multiple results in **bold** if confidence intervals overlap.

Table IV reports the AP performance of the various streaming algorithms on a large multivariate time series dataset, SMD, for various window sizes. Both constant and percentage-

TABLE V
AVERAGE INSERTION TIME (MS) ON SMALL DATASETS FOR A WINDOW SIZE OF 1%

Dataset	STREAMRHF	XSTREAM	RRCF	HST
abalone	$\mathbf{0.92 \pm 0.019}$	2.17 ± 0.474	30.028 ± 0.399	288.479 ± 1.051
annthyroid	$\mathbf{0.991 \pm 0.016}$	2.031 ± 0.448	49.385 ± 0.572	83.541 ± 0.454
kdd_ftp	$\mathbf{0.912 \pm 0.011}$	1.412 ± 0.304	30.805 ± 0.418	110.92 ± 0.37
magicgamma	$\mathbf{1.077 \pm 0.03}$	1.94 ± 0.434	58.881 ± 1.905	36.701 ± 0.188
mammography	$\mathbf{0.977 \pm 0.013}$	1.738 ± 0.392	41.352 ± 0.875	55.319 ± 0.134
mnist	7.581 ± 0.271	$\mathbf{4.604 \pm 1.064}$	66.781 ± 3.169	84.071 ± 0.42
musk	13.04 ± 0.533	$\mathbf{5.297 \pm 1.15}$	65.808 ± 2.631	202.636 ± 0.843
satellite	$\mathbf{2.538 \pm 0.067}$	$\mathbf{2.715 \pm 0.585}$	50.106 ± 0.779	116.514 ± 0.463
satimages	$\mathbf{2.441 \pm 0.018}$	$\mathbf{2.615 \pm 0.569}$	50.969 ± 0.669	102.838 ± 0.53
spambase	4.839 ± 0.206	$\mathbf{2.182 \pm 0.473}$	67.979 ± 2.389	129.742 ± 0.368
thyroid	$\mathbf{1.039 \pm 0.012}$	2.218 ± 0.479	40.759 ± 0.593	148.494 ± 0.617
mean	$\mathbf{3.305 \pm 0.69}$	$\mathbf{2.629 \pm 0.266}$	50.259 ± 2.468	123.569 ± 12.614
median	$\mathbf{1.084}$	2.412	49.985	110.812

Values are averages over 10 runs with 0.95 confidence interval. Best results are reported in **bold**. Multiple results in **bold** if confidence intervals overlap.

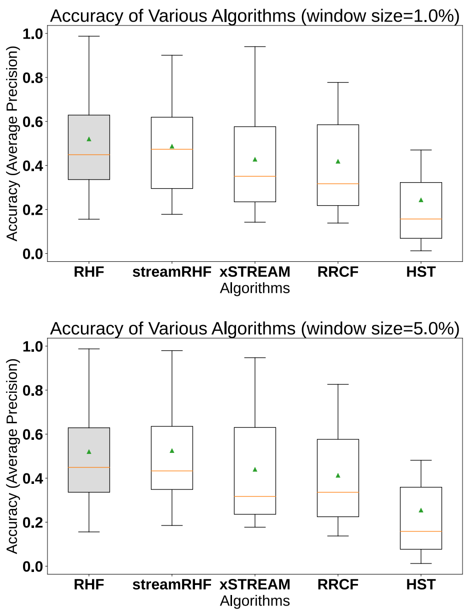

Fig. 2. Average precision on "small" datasets for two different window sizes. RHF is greyed out as it as a batch algorithm with batch scoring. The orange horizontal segments (resp. the green triangle points) represent the medians (resp. the means) over the datasets.

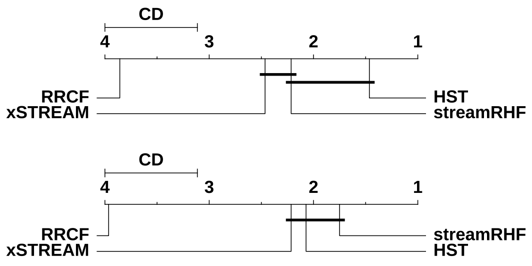

Fig. 3. Average rank of algorithms with the Nemenyi test on SMD entities for a window size of 256 (top) and 4096 (bottom). Groups of methods that are not significantly different (p = 0.05) are connected. CD is the critical distance required to reject equivalence.

based window sizes were used. The results show that STREAMRHF performs best with larger windows since it matches or outperforms all algorithms when the window contains at least 2048 instances. Moreover, it consistently and largely outperforms RRCF, a state-of-the-art algorithm. Figure 3 shows a statistical test that ranks how our algorithm compares to the others over all 28 entities of SMD when a window size of 256 and 4096 are used. The advantages of STREAMRHF on temporal data are more apparent with a larger window size of 4096, where it ranks first. However, even on a small window of 256 instances, our algorithm is not significantly different from two state-of-the-art solutions. Along with the XSTREAM and HST, it consistently outperforms RRCF.

Computational complexity results are tabulated in Table V and depicted in Figure 4 as boxplots. It can be seen that RRCF and HST have a significant computational overhead compared to XSTREAM and STREAMRHF, that are up to two order magnitude faster than the former. In particular, close comparison of STREAMRHF and XSTREAM reveals that the former has a consistently lower median insertion time (see Table V) but a wider inter-quantile (see Figure 4) with respect to the the latter. Otherwise stated, STREAMRHF and XSTREAM have comparable mean insertion time (around 3ms). Additionally, STREAMRHF is faster than XSTREAM in more than 50% of the cases (around 1ms) and is slower than XSTREAM in 25% of the cases (around 10ms). Overall, as far as computational speed is concerned, both STREAMRHF and XSTREAM appear to exhibit lightweight operation.

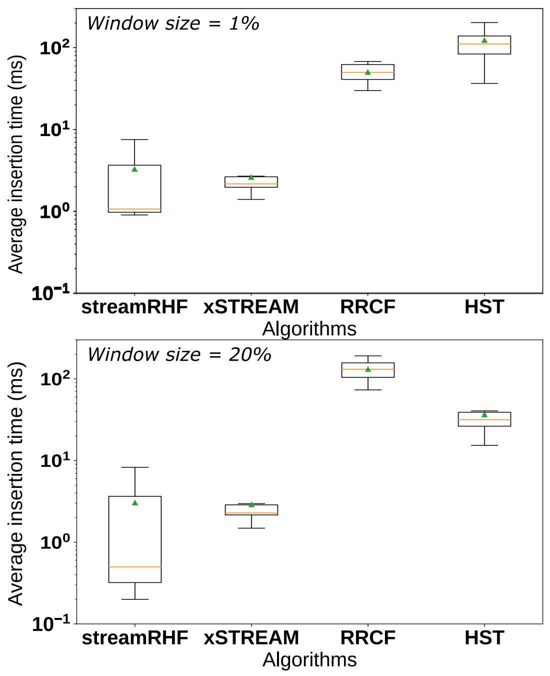

Fig. 4. Average insertion time of algorithms (ms) on "small" datasets with different window sizes. The orange horizontal segments (resp. the green triangle points) represent the medians (resp. the means) over the datasets.

V. Conclusions and Future Work

This paper introduces stream random histogram forest (STREAMRHF), a one-pass streaming algorithm for unsupervised anomaly detection. STREAMRHF maintains a sliding window, where only the most recent data instances are retained: the point-by-point reactivity and the use of a sliding window allows STREAMRHF to efficiently deal with changes in data distribution, i.e., concept drift.

We conducted an extensive experimental evaluation on a diverse set of datasets containing more than 600k instances from a various set of real-world problems. Our evaluation shows that: (i) STREAMRHF delivers comparable average precision with its batch counterpart RHF, despite the requirements of the stream settings (e.g., one-pass, not keeping the whole stream); and (ii) STREAMRHF outperforms state-of-the-art approaches in terms of average precision, while being among the fastest in terms of computational complexity.

For future work, it would be interesting to try a different windowed approach by deleting the oldest instance as soon as the newest one is inserted. This would require a deletion algorithm that is complementary to the one used for insertion. This new approach would remove an instance from a Random Histogram Tree and update the kurtosis values using decremental kurtosis formulas derived from the incremental ones. The model should then be even more accurate at each new instance and this should also optimize our algorithm, which would no longer rebuild from scratch at the end of the two windows.

References

[1] A. Putina, M. Sozio, D. Rossi, and J. M. Navarro, “Random histogram forest for unsupervised anomaly detection,” in *International Conference on Data Mining (ICDM)*. IEEE, 2020, pp. 1226–1231.

[2] C. Rudin, “Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead,” *Nat. Mach. Intell.*, vol. 1, no. 5, pp. 206–215, 2019.

[3] M. U. Togbe, Y. Chabchoub, A. Boly, M. Barry, R. Chiky, and M. Bahri, “Anomalies detection using isolation in concept-drifting data streams,” *Computers*, vol. 10, no. 1, p. 13, 2021.

[4] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander, “LOF: identifying density-based local outliers,” in *international Conference on Management of Data ACM SIGMOD*, 2000, pp. 93–104.

[5] F. T. Liu, K. M. Ting, and Z.-H. Zhou, “Isolation forest,” in *2008 eighth ieee international conference on data mining*. IEEE, 2008, pp. 413–422.

[6] B. Schölkopf, R. C. Williamson, A. J. Smola, J. Shawe-Taylor, J. C. Platt *et al.*, “Support vector method for novelty detection.” in *Neural Information Processing Systems (NIPS)*, vol. 12, 1999, pp. 582–588.

[7] S. C. Tan, K. M. Ting, and T. F. Liu, “Fast anomaly detection for streaming data,” in *International Joint Conference on Artificial Intelligence*, 2011.

[8] C. C. Aggarwal and S. Y. Philip, “A survey of synopsis construction in data streams,” in *Data Streams*. Springer, 2007, pp. 169–207.

[9] W. Ng and M. Dash, “Discovery of frequent patterns in transactional data streams,” in *Transactions on large-scale data-and knowledge-centered systems II*. Springer, 2010, pp. 1–30.

[10] M. Bahri, A. Bifet, S. Maniu, and H. M. Gomes, “Survey on feature transformation techniques for data streams,” in *International Joint Conferences on Artificial Intelligence (IJCAI)*, 2020.

[11] E. Manzoor, H. Lamba, and L. Akoglu, “xstream: Outlier detection in feature-evolving data streams,” in *ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2018, pp. 1963–1972.

[12] S. Guha, N. Mishra, G. Roy, and O. Schrijvers, “Robust random cut forest based anomaly detection on streams,” in *International Conference on Machine Learning (ICML)*, 2016, pp. 2712–2721.

[13] M. Bahri, F. Salutari, A. Putina, and M. Sozio, “AutoML: state of the art with a focus on anomaly detection, challenges, and research directions,” *International Journal of Data Science and Analytics*, pp. 1–14, 2022.

[14] M. Bartos, A. Mullapudi, and S. Troutman, “RRCF: Implementation of the Robust Random Cut Forest algorithm for anomaly detection on streams,” *The Journal of Open Source Software*, vol. 4, no. 35, p. 1336, 2019.

[15] T. Pevnỳ, “Loda: Lightweight on-line detector of anomalies,” *Machine Learning*, vol. 102, no. 2, pp. 275–304, 2016.

[16] S. Hettich and S. D. Bay, “UCI KDD archive,” 1999. [Online]. Available: http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html

[17] D. Dua and C. Graff, “UCI machine learning repository,” 2017. [Online]. Available: http://archive.ics.uci.edu

[18] S. Rayana, “ODDS library,” 2016. [Online]. Available: http://odds.cs.stonybrook.edu

[19] Y. Su, Y. Zhao, C. Niu, R. Liu, W. Sun, and D. Pei, “Server Machine Dataset [https://github.com/NetManAIOps/OmniAnomaly],” 2019.

[20] J. Davis and M. Goadrich, “The relationship between precision-recall and roc curves,” in *International Conference on Machine Learning (ICML)*, ser. ICML ’06, New York, NY, USA, 2006, p. 233–240.