STREAMRHF: A review and a CapyMOA implementation

Giovanni BENEDETTI DA ROSA
Cristian Alejandro CHÁVEZ BECERRA

Master 2 Data Science
Institut Polytechnique de Paris-École Polytechnique
Data Streaming
giovanni.benedetti-da-rosa@polytechnique.edu
cristian.chavez-becerra@polytechnique.edu
https://github.com/AlejandroUN/Stream-Random-Histogram-Forest

Abstract. Data streaming algorithms must address the challenges of limited memory and real-time processing requirements, which are prevalent in many modern applications such as anomaly detection, fraud prevention, and network monitoring. Traditional anomaly detection methods are often ill-suited for streaming data due to their reliance on batch processing and the difficulty of obtaining labeled datasets. In this work, we implement and evaluate the StreamRHF algorithm (a tree-based, unsupervised anomaly detection method for data streams) within the CapyMOA Python library. StreamRHF leverages kurtosis-based splits to construct Random Histogram Forests incrementally, allowing it to adapt efficiently to concept drifts in streaming environments.

In this report we review the already existing methods and we replicate the results of the original paper, demonstrating competitive performance across various datasets comparing with CapyMOA built-in methods. Additionally, we extend the algorithm's capabilities by normalizing its anomaly scores to integrate with CapyMOA's framework. Experimental results show that StreamRHF outperforms or matches state-of-the-art methods such as HalfSpaceTrees and Online Isolation Forests in terms of anomaly detection metrics. However, the method's computational complexity presents challenges, particularly regarding runtime. We attribute this to the fact the code was built in Python, and that we used larger Forests than the other methods. Github repository can be found in`https://github.com/AlejandroUN/Stream-Random-Histogram-Forest`

1 Introduction

In contrast with batch learning algorithms, data streaming algorithms need to satisfy strict memory requirements and faster processing times. Since several applications work with a large volume of data that are updated in real-time, data streaming has several applications such as anomaly detection, fraud detection, network monitoring, and real-time recommendation systems. Anomaly detection consists of identifying interesting or rare events that deviate from normality and can be applied in fields like cybersecurity, finance, and industrial systems.

The great majority of anomaly detection algorithms are not adapted to the requirements of data streaming algorithms [2]. In this sense, it is valuable to mention that, as in anomaly detection, it is hard to obtain labeled datasets, supervised approaches may be not usefull.

The goal of this project is to implement the algorithm presented in the paper *STREAMRHF: A Tree-Based Unsupervised Anomaly Detection for Data Streams*[2] within the open-source Python library, CapyMOA.

The StreamRHF is an unsupervised anomaly detection, based on the ideas of Random Forest Histograms(RHF)[3], where the feature splits are constructed based on the kurtosis score of every feature. The tree-based approach implemented in the paper incrementally computes the random trees as new data streaming points arrive in the model, making the model efficient and more explainable.

Integrating this state-of-the-art algorithm into CapyMOA can significantly benefit the community. CapyMOA is a Python-based tool specifically designed for data stream analysis, offering seamless integration with MOA, PyTorch, and scikit-learn [1].

In the following sections, we will explore the theoretical concepts and key ideas underlying the algorithm. Furthermore, we will replicate key experiments from the original paper and compare our implementation with the existing anomaly detection methods in CapyMOA to validate its accuracy and demonstrate its performance.

2 Methods

First, we will review the theoretical concepts underlying the StreamRHF algorithm. In the following section, we will present the batch learning method, RHF(Random Histogram Forests), which serves as the foundation for the algorithm. Next, we present how to adapt the batch learning method to its streaming version, StreamRHF.

2.1 Random Histogram Forests(RHF)

RHF is an ensemble model that splits he input points using several splits drawn at random. Intuitively, points that end up in a relatively large group are less likely to be anomalies. By iterating such a process multiple times, we can measure how likely a point is an anomaly [3].

To achieve this, the data is split across different bins by means of a Random Histogram Tree (RHT). Each one of the t trees with height h is built by recursively splitting the whole dataset: each splitting decision is done selecting an attribute a to split according to its kurtosis score and a split value randomly selected from its support.

Kurtosis Split Criterion As defined by Putina et al. [3], the Random Histogram Tree can be constructed following the procedure:

Given a dataset $X \in \mathbb{R}^{n \times d}$, where n is the number of instances and d is the number of features or attributes, a Random Histogram Tree (RHT) is a binary tree in which a node Q_i is either: an internal node with exactly two children, or a leaf node with no children.

In the case of an internal node, the node splits the dataset into a left branch X_{Q_L} and a right branch X_{Q_R}, according to the *kurtosis split* criterion. A RHT contains at most $\eta = 2^h$ leaves, where h is the maximum height of the tree.

Given a random variable X , the kurtosis score is defined[4] as:

$$K[X] = \mathbb{E}\left[\left(\frac{X-\mu}{\sigma}\right)^4\right] = \frac{\mathbb{E}\left[(X-\mu)^4\right]}{\mathbb{E}\left[(X-\mu)^2\right]^2} = \frac{\mu_4}{\sigma^4}, \tag{1}$$

where μ^4 is the fourth central moment and σ the standard deviation, measures the tailedness of X.

In this sense, the kurtosis score measures the "tailedness" of a distribution, since variables near the peak have a negligible contribution, while regions far from the peak, i.e., outliers, contribute more significantly to the measure. As cited in [3], Moors [4] described kurtosis as a measure of dispersion and explained that high kurtosis values result from either: (i) occasional values far from the mean in a distribution with most of its probability mass concentrated around the mean, or (ii) significant probability mass located in the distribution's tails. In the figure 1, it's possible to visualize 4 features extracted from the dataset *annthyroid*. As discussed before, we can check that the features with with heavier tails and consequently higher kurtosis score are more likely to contain anomalous instances.

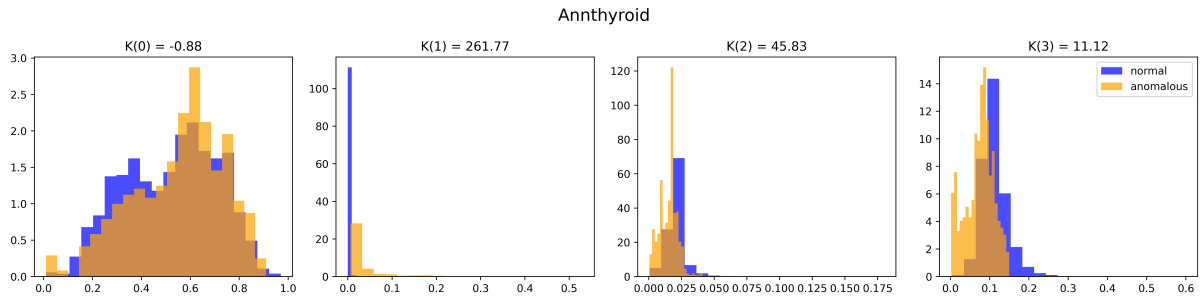

Fig. 1: PDF of 4 features extracted from Annthyroid dataset

In order to prevent that small variations in the split and avoiding that the algorithm struggles with *0-Kusrtosis* features, we can define a *Kurtosis split*[2] as :

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

$$K_s = \sum_{a=0}^{d} \log\big(K(X_a) + 1\big). \tag{2}$$

2) A random number r is chosen from a uniform distribution:

$$r \sim \text{Uniform}([0, K_s)). \tag{3}$$

3) The split attribute a_s is selected based on this r value as:

$$a_s = \arg\min \left(k \in [0, d] \;\middle|\; \sum_{a=0}^{k} \log\big(K(X_a) + 1\big) > r \right), \tag{4}$$

while the split value v_a is chosen randomly (uniformly) within the observed range of the selected feature. The trees are split until either the maximum height is reached or the kurtosis of all attributes becomes zero, indicating that the remaining instances are duplicates.

Anomaly Score By definition, a leaf is a node without any children, containing at least one instance. Given that S_l represents the number of instances in each leaf, each instance i is stored in a tree T, it's possible to write the *information content*[3]:

$$w_T(i) = \log\left(\frac{1}{P_{l_i}}\right) \tag{5}$$

where P_l is the probability associated with leaf l. Given a leaf l, we denote $S(l)$ as the set of distinct instances associated with l. We also define $P_l := \frac{|S(l)|}{n}$, which represents the probability that an instance is associated with leaf l.

The total score of i, denoted as w_i, is calculated as the sum of the scores across all t trees:

$$w_i = \sum_{T} w_T(I). \tag{6}$$

The Information Content, also defined as Shannon's information, measures the level of "surprise" associated with an event.

In algorithm 1 it's possible to see the pseudocode of the algorithm, to build a Random Histogram Forest, using the Anomaly score algorithm 2 procedure detailed [3].

Algorithm 1 RHF(X, nd, h)

Input: Dataset X, node depth nd, and max height h
Output: RHF

if $nd \geq h$ or $|X| = 1$ **then**
  return Leaf$\{S(Q)\}$
else
  Let A be the set of attributes
  Select the attribute to split a_s according to kurtosis
  Split described in 2, 3, 4
  Select a random split value a_v in the min and max support of a_s in X
  $X_l \leftarrow \text{filter}(X \mid a_s < a_v)$
  $X_r \leftarrow \text{filter}(X \mid a_s \geq a_v)$
  return Node{
    value = a_v
    attribute = a_s
    left = RHT$(X_l, nd + 1, h)$
    right = RHT$(X_r, nd + 1, h)$
  }
end if

Algorithm 2 Score(x, node)

Input: Instance $x \in X$ and an RHT node
Output: Anomaly score given RHT

if node is a `Leaf` **then**
  $P \leftarrow \frac{\text{node}\{S(Q)\}}{|X|}$
  return $\log\left(\frac{1}{P}\right)$
else
  $a \leftarrow$ node.attribute
  $v \leftarrow$ node.value
  if $x_a < v$ **then**
    return `Score`$(x,$ node.left$)$
  else
    return `Score`$(x,$ node.right$)$
  end if
end if

As can be noticed, the algorithm proposed by Putina et al. [3] is a batch algorithm. For this reason, in the next section, we will present its stream version.

2.2 StreamRHF

When handle with data streams the distribution of data is not time invariant, leading to concept drifts, meaning that the model may no longer accurately reflect the distribution of data. To address this, we need to update the model so it take into account the new data that arrives.

A naive solution to address this issue would be to compute a new RHF each time new data arrives; however, this approach is computationally expensive and impractical for data streams. An alternative solution could involve rebuilding the subtree whenever the kurtosis deviates significantly from the value computed during the last rebuild. However, this method requires defining and tuning an appropriate threshold for the kurtosis deviation, which adds complexity.

The StreamRHF algorithm offers a parameter-free solution to the stream case, to rebuild the RHF ads close as possible to the batch case. 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 10 and 6. When a new instance x arrives.

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. If the value of the splitting attribute in x is less than or equal to the splitting value, we proceed recursively through the left subtree; otherwise, we move to the right subtree. This process continues recursively until a leaf node is reached, where the score of x is computed using Equation 10. This approach enables the selective rebuilding of only a partial subtree, instead of a full tree.

As can be seen in equation 4, the splitting attribute is determined by the random value r. So, to efficiently decide when a subtree needs to be rebuilt, a random value r is initialized for each node in every tree and maintained throughout the entire data stream.

Put into equations what was said above, let a_{current} be the existing split attribute in the model. The splitting attribute is then selected as follows:

$$a_{\text{new}} = \arg\min \left(k \;\middle|\; \sum_{a=0}^{k} \log\left(K(X'_a) + 1\right) > r \right) \tag{7}$$

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

Concept Drift Concept drift in data streams is a critical challenge. In this context, concept drift describes changes in their underlying distribution. In real-world applications, data distributions are often non-stationary, causing models that were once effective to become non-representative over time, when next data arrive.

To deal with Concept Drift, the authors proposed a window approach to update the random seed(r), when new data arrives. There are 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 the algorithm described in the previous session. When the current window is filled with n values the reference window is overwritten by the current window while this last one will be emptied. The model will be rebuild and be used until the a new current window is fixed.

The full algorithm as expressed in the paper[2] is exposed in Algorithm and and

Algorithm 3 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`

if node is not `Leaf` **then**
  $kvalues \leftarrow$ `kurtosis(node.data` $\cup i$`)` ▷ same function as in the batch, but the seed of every node is initialized using values of z
  $a_{split} \leftarrow$ `get-attribute(`$kvalues, z$`[node.id])`
  if `node.attribute` $\neq a_{split}$ **then**
    return `RHT-build(node.data` $\cup i$, `node.height)`
  end if
  if i`[node.attribute]` $\leq$ `node.value` **then**
    `node.left` $\leftarrow$ `Insert(node.left,` i, h, z`)`
  else
    `node.right` $\leftarrow$ `Insert(node.right,` i, h, z`)`
  end if
else
  if `node.height` $= h$ **then**
    `node.data` $\leftarrow$ `node.data` $\cup i$

Algorithm 4 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

scores $\leftarrow$ []
for i in $0..2^h - 1$ **do**
  $z[i][j] \leftarrow$ `RandomSeed()` ▷ set seeds, one per node per tree, assuming complete tree
end for
forest $\leftarrow$ `RHF-build(`$X[0:n], h, t, z$`)`
scores[0:n] $\leftarrow$ `RHF-score(forest,` $X[0:n]$`)`
for i in $X[n+1$: `end`$]$ **do**
  for all tree in forest **do**
    tree $\leftarrow$ `Insert(tree,` i, z`[tree.id])`
  end for
  scores[i] $\leftarrow$ `RHF-score(forest,` i`)`
  if $i\%n == 0$ **then** ▷ reference window is overwritten by the observations in the current window
    forest $\leftarrow$ `RHF-build(`$X[i-n:i], h, t$`)`
  end if
end for

Time and Space Complexity Since for every data point that arrives we need to store, for every tree, all points of the window, StreamRHF requires $\mathcal{O}(wdt)$ space complexity, where w is the size of considered window, d is the number of features, and t is the number of trees in the forest. The initialization step runs in $\mathcal{O}(wdth)$, as, in the worst case, all points in the sliding window are processed at every tree level. Updates can also take $\mathcal{O}(wdth)$ if the entire tree is rebuilt, but this is rare. Typically, only smaller subtrees with height $h < \bar{h}$ are rebuilt, reducing the update time to $\mathcal{O}(wdt\bar{h})$ [2].

3 Experimental evaluation

To evaluate our implementation, we replicated similar experiments on the same datasets used in the original paper [2]. Additionally, we compared our results with the built-in methods available in CapyMOA, using the same datasets. A small demo was also conducted to demonstrate the compatibility with the library.

3.1 Experimental Settings

Datasets We used the same datasets that were used in the original paper[2], they consisted of 12 of the 17 datasets, including *small* and *large* datasets according to the authors classification.

Environment We built all our code in Python 3, utilizing important libraries such as Numpy and `multiprocessing` to parallelize the creation of trees using a long-lived process pool. The experiments were conducted on a server running *Ubuntu 24.04.1 LTS*, equipped with an *Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz*, *62 GiB of RAM*, *20 physical cores*, and *40 threads*.

Parameters We utilized all the built-in methods available for anomaly detection in CapyMOA with their default parameter values. The methods and their configurations are as follows:

- **HalfSpaceTrees**[5]: `window_size` = 100, `number_of_trees` = 25, `max_depth` = 15, `anomaly_threshold` = 0.5.
- **Online Isolation Forest**: `num_trees` = 32, `max_leaf_samples` = 32, `growth_criterion` = 'adaptive', `subsample` = 1.0, `window_size` = 2048, `branching_factor` = 2, `split` = 'axisparallel', `n_jobs` = 1. This method is based on the work Leveni et al. [6].
- **Autoencoder**: `hidden_layer` = 2, `learning_rate` = 0.5, `threshold` = 0.6. This method is based on the work by Moulton et al. [7].

Our implementation of **StreamRHF** was run outside CapyMOA to parallelize the creation of trees and speed up the experiments. For the hyperparameters, we set $T = 100$ and $H = 5$, as in the original paper [2].

Metrics In anomaly detection, the proportion of normal to anomalous instances is often highly imbalanced, which must be considered when evaluating the validity of a method. It has been shown that for fixed outlier fractions AUCPR can be provide robust evaluations. In contrast, for variable outlier fractions, ROC AUC, as it is unaffected by changes in outlier prevalence and provides consistent evaluations [8].

To handle this situation, we measure the Average Precision (AP) of the Precision-Recall Area Under the Curve (without interpolation), as defined in [3]:

$$AP = \sum_n (R_n - R_{n-1}) P_n, \tag{8}$$

where

$$P_n = \frac{TP}{TP + FP} \quad \text{and} \quad R_n = \frac{TP}{TP + FN},$$

represents the precision and recall, with TP, FP, and FN representing true positives, false positives, and false negatives, respectively.

Additionally, we computed the ROC AUC, which can be defined as:

$$AUC = \sum_{i=1}^{N} (FPR_i - FPR_{i-1}) \cdot TPR_i, \tag{9}$$

where TPR is the True Positive Rate, FPR is the False Positive Rate, and the integral is taken over all possible thresholds.

3.2 Implementation on CapyMOA

For the CapyMOA implementation of the StreamRHF algorithm, a class called `stream_rhf.py` was created in the `src/capymoa/anomaly` folder.

Next, it was added to the `__init__.py` file in the `src/capymoa/anomaly` folder, referencing the class as `StreamRHF`.

Implementation Details: For the implementation, we created all the classes from scratch to be sure of the methods and procedures being carried out. We created 3 classes inside the `stream_rhf.py` file:

Node: This class represents the nodes of the RandomHistogramForest, it has important functions to compute the kurtosis value (used later to divide the datapoints in a node) and to choose the feature to separate the data.

RandomHistogramForest: This class is in charge of initializing and updating the Random Histogram Forest (based on the paper `https://nonsns.github.io/paper/rossi20icdm.pdf`) as well as giving an anomaly score to the datapoints.

Since CapyMOA requires an anomaly score between 0 and 1 and the paper anomaly score gives values way bigger than one depending on the number of instances we normalized the values in the following way:

First we calculate the minimum and the maximum possible anomaly score an instance may have. Since we use the following formula to calculate the anomaly score of an instance

$$w_T(i) = \log\left(\frac{1}{P_{l_i}}\right) = \log\left(\frac{n}{|S(l)|}\right) \tag{10}$$

the maximum value is obtained when there is only one instance in the leaf, so that, the maximum anomaly score is:

$$\text{max\_score} = \log\left(\frac{n}{1}\right) = \log(n). \tag{11}$$

The minimum anomaly score occurs when all instances are in the same leaf, so that, the minimum anomaly score is:

$$\text{min\_score} = \log\left(\frac{n}{n}\right) = \log(1) = 0. \tag{12}$$

To normalize the anomaly score $w_T(i)$ to the range $[0, 1]$, we use the following formula:

$$\text{normalized\_score} = \frac{w_T(i) - \text{min\_score}}{\text{max\_score} - \text{min\_score}}, \tag{13}$$

StreamRHF: This class is the implementation of the StreamRHF method and uses the methods of the previously mentioned classes. This class is the one used by CapyMOA and has the methods of the anomaly detectors in CapyMOA for: Training the model given an instance. Giving an anomaly score to an instance (the closer the value is to 0, the more likely it is an anomaly). Predicting an instance: This method uses a threshold (by default 0.5) to decide whether an instance is classified as an anomaly or not. If the score is lower than the threshold, then it is classified as an anomaly.

In the figure 2 we can see the succesful integration of our implementation in the CapyMOA library being able to plot the its performances along with the other models using CapyMOA evaluators and plots

Tests: For testing, the `StreamRHF` class was imported into the `test_anomaly_detectors.py` file. By default, the model is initialized with the following hyperparameter values:`max_height=5`, `num_trees=100` and `window_size=20`

The AUC value that `StreamRHF` achieves on the `ElectricityTiny` dataset with the default hyperparameters is approximately `0,505`. With these values, the runtime is approximately `2 hours`.

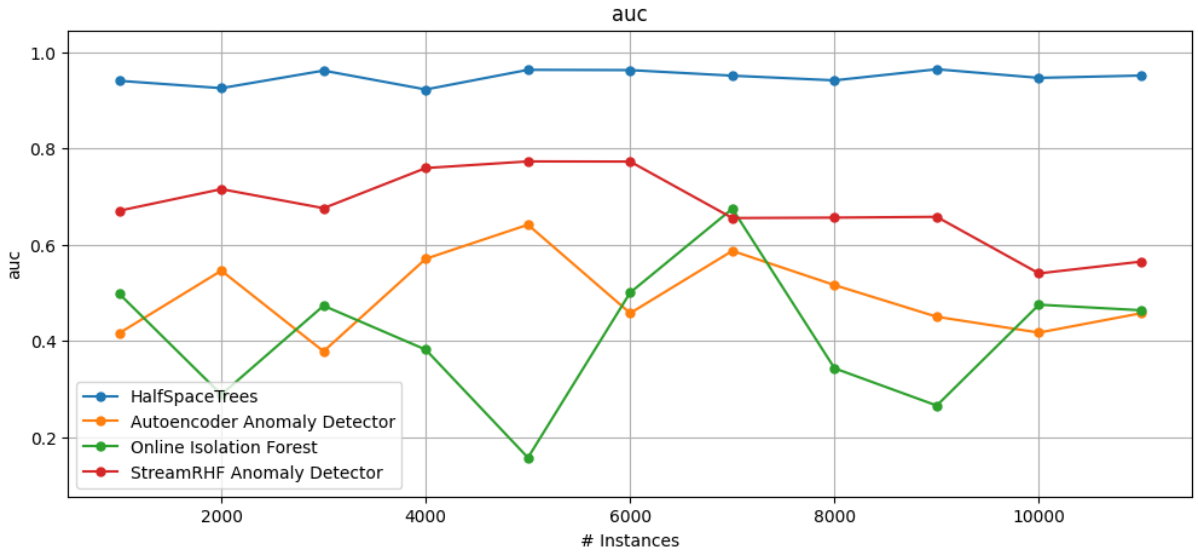

Fig. 2: Plot of CapyMOA comparison

Documentation: Additionally, the model, class, and its methods were documented following the CapyMOA documentation guidelines. The documentation uses the `sphinx/reStructuredText` style adopted by CapyMOA. To generate the documentation, the following command was executed:

```
python -m invoke docs.build -i
```

This command generates the documentation page for `StreamRHF`, including its description and usage.

3.3 Results

In the table 1, we expose the AP of the models described in the previous sections for all datasets. First, we observe that, in general, the average precision (AP) of our implementation across the datasets is very similar to the values reported in the original paper [2]. The mean AP of our results across the datasets closely matches the reported value of 0.527 from the original paper.

While there are some fluctuations in the AP for certain datasets—for example, the abalone dataset shows a deviation of around 5% compared to the original experiment—in other cases, such as the mammography and musk datasets, the values are nearly identical to those reported. These results demonstrate the consistency of our implementation with the original work while accounting for potential variations in experimental setups or data handling. We believe that can be a difference in the spli critera as defined in the original RHF paper and in the streamRHF.

We observe that, with respect to the AP metric, StreamRHF generally outperforms the default configurations of the built-in CapyMOA models. However, in the case of very small datasets, such as the abalone dataset, HalfSpaceTrees surpasses StreamRHF. This comparison was not included in the original paper.

In the table 2, StreamRHF doesn't always outperform other CapyMOA models in terms of AUC, but it delivers competitive results, often close to the best model, HalfSpaceTrees. Its average and median AUC values are still higher, showing its overall reliability.

While the results in terms of metrics are good for StreamRHF, it has a notable drawback: the complexity of the algorithm. It takes significantly longer than other models, which can be attributed to several factors. First, StreamRHF uses 100 trees in all iterations, a much higher number compared to OnlineIsolationForest (32 trees) and HalfSpaceTrees (25 trees by default). Second, the superior performance of HalfSpaceTrees in some cases could also be attributed to its implementation using the MOA framework [9], which is built in Java. In this sense, the original paper was implemented in Cython, which also explains the difference from ours results.

Nonetheless, there are ways to reduce the running time of streamrhf. 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.

Dataset	StreamRHF	Autoencoder	HalfSpaceTrees	OnlineIsolationForest
abalone	0.237 ± 0.024	0.116	**0.531**	0.033
annthyroid	0.396 ± 0.019	0.086	**0.577**	0.079
kdd_ftp	0.280 ± 0.008	0.157	**0.439**	0.196
magicgamma	**0.623 ± 0.008**	0.352	0.272	0.287
mammography	**0.219 ± 0.021**	0.157	0.184	0.049
mnist	**0.303 ± 0.006**	0.092	0.117	0.085
musk	**0.771 ± 0.061**	0.032	0.021	0.495
satellite	**0.593 ± 0.012**	0.015	0.317	0.518
satimages	**0.856 ± 0.020**	0.012	0.256	0.012
shuttle_odds	**0.858 ± 0.044**	0.072	0.548	0.040
spambase	0.478 ± 0.038	0.507	0.282	**0.568**
thyroid	**0.544 ± 0.031**	0.103	0.559	0.014
Mean	**0.513 ± 0.024**	0.142	0.342	0.198
Median	**0.511**	0.098	0.300	0.082

Table 1: AP scores for different datasets and models.

Notes: Window size 1%.Values are the average and 0.95 confidence interval over 10 runs for small datasets, while for the large we used runs. The best results of algorithms are in **bold**. There was a problem in setting the *seed*, that's why there is no variability for Capymoa methods.

Dataset	StreamRHF	Autoencoder	HalfSpaceTrees	OnlineIsolationForest
abalone	0.862 ± 0.005	0.855	**0.963**	0.318
annthyroid	0.865 ± 0.006	0.534	**0.974**	0.516
kdd_ftp	0.423 ± 0.022	0.124	**0.662**	0.350
magicgamma	**0.701 ± 0.010**	0.500	0.382	0.396
mammography	0.858 ± 0.004	0.766	**0.912**	0.749
mnist	**0.809 ± 0.005**	0.500	0.558	0.499
musk	**0.980 ± 0.007**	0.500	0.026	0.612
satellite	0.939 ± 0.009	0.500	**0.975**	0.541
satimages	**0.989 ± 0.003**	0.500	0.965	0.494
shuttle_odds	**0.994 ± 0.003**	0.500	0.899	0.101
spambase	0.636 ± 0.034	**0.678**	0.271	0.496
thyroid	0.977 ± 0.001	0.823	**0.979**	0.168
Mean	**0.839 ± 0.011**	0.556	0.725	0.423
Median	**0.862**	0.500	0.662	0.396

Table 2: AUC scores for different datasets and models.

Notes: Window size 1%. Values are the average and 0.95 confidence interval over 10 runs. The best results of algorithms are in **bold**. There was a problem in setting the *seed*, that's why there is no variability for Capymoa methods.

Indeed, in most of the cases, we rebuild only relatively small subtrees when doing an insertion with height $\bar{h} < h$, which gives a running time of $O(wdt\bar{h})$ for an update. Reducing the number of trees may decrease the robustness of the ensemble predictions, the core of StreamRHF. Reducing the maximum height of the trees may reduce the granularity of the trees reducing the ability to distinguish the anomalies from normal points

CapyMOA evaluation procedures: In the original paper, the procedure to evaluate the model the authors did for every instance was to first insert the instance and then score the instance. This means that for both a normal and anomaly instance it would reduce its anomaly score since when trying to score the instance is going to arrive to the same leaf where the same isntance was previously inserted.

Dataset	StreamRHF	Autoencoder	HalfSpaceTrees	OnlineIsolationForest
abalone	117.338 ± 1.076	0.310 ± 0.015	**0.098 ± 0.026**	5.642 ± 0.044
annthyroid	122.116 ± 1.671	0.303 ± 0.001	**0.032 ± 0.005**	10.633 ± 0.023
kdd_ftp	112.115 ± 1.050	0.298 ± 0.000	**0.032 ± 0.001**	10.603 ± 0.020
magicgamma	202.003 ± 7.166	0.305 ± 0.001	**0.033 ± 0.010**	13.816 ± 0.032
mammography	134.445 ± 2.414	0.304 ± 0.003	**0.031 ± 0.003**	13.813 ± 0.018
mnist	398.449 ± 5.247	0.370 ± 0.001	**0.032 ± 0.000**	13.300 ± 0.023
musk	547.362 ± 12.781	0.422 ± 0.002	**0.045 ± 0.024**	8.237 ± 0.040
satellite	206.419 ± 4.093	0.324 ± 0.000	**0.032 ± 0.000**	9.457 ± 0.026
satimages	205.343 ± 6.415	0.323 ± 0.001	**0.032 ± 0.000**	11.490 ± 0.033
shuttle_odds	189.577 ± 6.058	0.304 ± 0.002	**0.029 ± 0.000**	12.556 ± 0.275
spambase	177.934 ± 2.605	0.338 ± 0.001	**0.032 ± 0.000**	7.627 ± 0.031
thyroid	112.401 ± 0.947	0.303 ± 0.001	**0.037 ± 0.007**	8.647 ± 0.021

Table 3: Insertion times (in ms) for different datasets and models, with the smallest values highlighted in bold.

Notes: Values are the average and 0.95 confidence interval over 10 runs.

However, since instance are not going to arrive to the same leaf, and since each leaf does not have the same number of instance it means that is going to reduce the anomaly score in different proportions. In CapyMOA however, in the examples given in the tutorial for anomaly detection models, the procedure to evaluate a model is to first get the score of the instance and then insert the instance in the model.

We decided to see how much may vary the results using these different evaluation procedures.

Dataset	Original Paper Procedure (AP)	CapyMOA Procedure (AP)
abalone	0.233 ± 0.035	0.123 ± 0.020
annthyroid	0.401 ± 0.035	0.354 ± 0.028
kdd_ftp	0.280 ± 0.011	0.169 ± 0.036
thyroid	0.548 ± 0.044	0.444 ± 0.017
Mean	0.365 ± 0.031	0.272 ± 0.012
Median	0.341	0.259

Table 4: Comparison of AP scores for different datasets using the original paper procedure and CapyMOA procedure.

Notes: Values are the average and 0.95 confidence interval over 5 runs.

As can be seen in table 4, the original paper procedure leads to better results for the AP score, at least, in the 4 datasets that we ran that are the smaller ones. In these 4 datasets the smallest variation between procedures corresponds to a variation of around 5% in the annthyroid dataset and biggest variation corresponds to a variation of around 11% for the kdd_ftp dataset.

4 Conclusion

StreamRHF is a novel method that shows promising results and competes with state-of-the-art data streaming models. Its ability to adapt to Drift Change depends especially on its windows size which can be adapted for each dataset to obtain a better overall performance. Our implementation in CapyMOA replicates the results of the original paper, demonstrating very similar results in detecting anomalies across the 12 of the original datasets of the paper. Compared to existing methods like HalfSpaceTrees, Online Isolation Forest, and Autoencoders in CapyMOA, StreamRHF achieves competitive performance in both datasets, the datasets from the original paper and the datasets from CapyMOA.

We also defined a maximum and minimum score that helped in the implementation of the method in CapyMOA to normalize the values in a scale between 0 and 1, necessary to the implementation of the method in the library.

The main problem of our implementation of this method in its complexity. We may reduce its time complexity by reducing the maximum height for the trees or by reducing the number of trees in the forest, however, these too approaches may reduce the performance of the method. Also, Python can be a bottleneck of the implementation when compared with other languages like Java. Future research could be done studying this trade-off and other possibilities to improve the computational complexity.

5 Code Availability

Our GitHub repository for the project can be found at:

`https://github.com/AlejandroUN/Stream-Random-Histogram-Forest`

The from-scratch implementation Python file of the algorithm StreamRHF can be found in the following link in our GitHub repository:

`https://github.com/AlejandroUN/Stream-Random-Histogram-Forest/blob/main/src/capymoa/anomaly/_stream_rhf.py`

The demo showing the implementation properly integrated with CapyMOA using CapyMOA models and plots to compare can be found in the following notebook:

`https://github.com/AlejandroUN/Stream-Random-Histogram-Forest/blob/main/src/capymoa/anomaly/streamrhf/notebook_presentation.ipynb`

References

1. Gomes, H. M., Bifet, A. (2024). Practical Machine Learning for Streaming Data. ACM Digital Library, 6418-6419. Available: https://doi.org/10.1145/3637528.3671442
2. Stefan Nesic, Andrian Putina, Maroua Bahri, Alexis Huet, Jose Manuel, et al.. STREAMRHF: Tree-Based Unsupervised Anomaly Detection for Data Streams. AICCSA 2022 - 19th ACS/IEEE International Conference on Computer Systems and Applications, Dec 2022, Abu Dhabi, United Arab Emirates. ffhal-03948938f
3. 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.
4. J. J. A. Moors, "The meaning of kurtosis: Darlington reexamined", The American Statistician, vol. 40, no. 4, pp. 283–284, 1986.
5. Swee Chuan Tan, Kai Ming Ting, and Tony Fei Liu. Fast anomaly detection for streaming data. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1469–1495, 2017.
6. Filippo Leveni, Guilherme Weigert Cassales, Bernhard Pfahringer, Albert Bifet, and Giacomo Boracchi. Online Isolation Forest. In *International Conference on Machine Learning (ICML), Proceedings of Machine Learning Research (PMLR)*, 2024.
7. Richard Hugh Moulton, Herna L. Viktor, Nathalie Japkowicz, and João Gama. Contextual one-class classification in data streams. *arXiv preprint arXiv:1907.04233*, 2019.
8. Minjae Ok, Simon Klüttermann, and Emmanuel Müller, "Exploring the Impact of Outlier Variability on Anomaly Detection Evaluation Metrics," arXiv preprint, arXiv:2409.15986v1 [cs.LG], 24 Sep 2024, License: CC BY 4.0.
9. Bifet, Albert, et al. Machine Learning for Data Streams: with Practical Examples in MOA. The MIT Press, 2018. DOI: https://doi.org/10.7551/mitpress/10654.001.0001.