xStream: Outlier Dete‘x’ion in Feature-Evolving Data Streams

Emaad Manzoor
H. John Heinz III College
Carnegie Mellon University
emaad@cmu.edu

Hemank Lamba
School of Computer Science
Carnegie Mellon University
hlamba@cs.cmu.edu

Leman Akoglu
H. John Heinz III College
Carnegie Mellon University
lakoglu@andrew.cmu.edu

ABSTRACT

This work addresses the outlier detection problem for feature-evolving streams, which has not been studied before. In this setting both (1) *data points may evolve*, with feature values changing, as well as (2) *feature space may evolve*, with newly-emerging features over time. This is notably different from row-streams, where points with *fixed* features arrive *one at a time*.

We propose a density-based ensemble outlier detector, called xStream, for this more extreme streaming setting which has the following key properties: (1) it is a constant-space and constant-time (per incoming update) algorithm, (2) it measures outlierness at multiple scales or granularities, it can handle (3*i*) high-dimensionality through distance-preserving projections, and (3*ii*) non-stationarity via $O(1)$-time model updates as the stream progresses. In addition, xStream can address the outlier detection problem for the (less general) disk-resident static as well as row-streaming settings.

We evaluate xStream rigorously on numerous real-life datasets in all three settings: static, row-stream, and feature-evolving stream. Experiments under static and row-streaming scenarios show that xStream is as competitive as state-of-the-art detectors and particularly effective in high-dimensions with noise. We also demonstrate that our solution is fast and accurate with modest space overhead for evolving streams, on which there exists no competition.

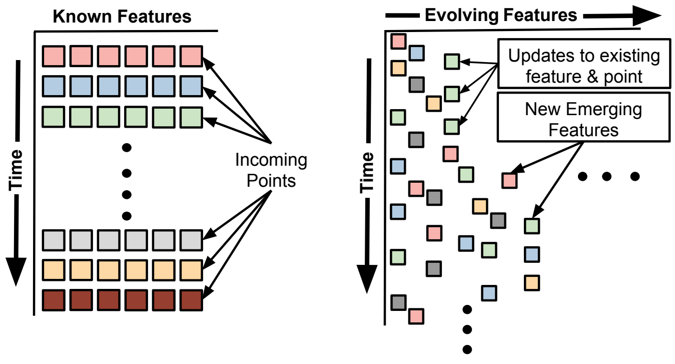
ACM Reference Format:
Emaad Manzoor, Hemank Lamba, and Leman Akoglu. 2018. xStream: Outlier Dete‘x’ion in Feature-Evolving Data Streams . In *KDD ’18: The 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, August 19–23, 2018, London, United Kingdom.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3219819.3220107

1 INTRODUCTION

How can we detect outlier data points in a stream, which “evolves” over time, that is a stream where not only new points arrive over time but also (*i*) the feature values of the points may change and (*ii*) previously unseen features may arrive?

Outlier detection has numerous key applications in finance, security, etc. and has been studied vastly [2]. All of existing work for point outlier detection can be categorized to one of two settings:

(1) static (d, n known): when the full $n \times d$ data matrix $\mathcal{D}$ is given, containing n points in d dimensions, and



(a) row-stream (b) *feature-evolving* stream

Figure 1: Row-streaming data [all prior work] vs. Feature-evolving data stream [this paper] (colors indicate unique *ids*): in (a) points arrive, get scored for outlierness, and discarded *one-at-a-time*, vs. in (b) not only new points (rows) arrive over the stream, but also (*i*) feature space evolves with newly-emerging features (columns), and (b) *id*’s evolve with feature-value updates arriving in an interleaved fashion. As such, (b) is inherently a more challenging setting.

(2) row-streaming (d known, n unknown): when d-dimensional rows of $\mathcal{D}$ arrive over time—one by one.

In this work we consider a (3)rd, new setting that has not been addressed before in any prior work: when indices of $\mathcal{D}$ arrive arbitrarily in time over a stream where *both n and d are unknown apriori.* Concretely, triplets of the form (id, f, δ) arrive over the stream, where id is the unique identifier for a data point, f is a unique feature name and δ is the amount of update (positive or negative) to the value of feature f for point id. Here both id and f may be previously unseen or seen, i.e. both new rows as well as new columns can be added to the data or values of existing indices can be updated over the stream. In this paper, we introduce the term *feature-evolving stream* (or evolving stream in short) to refer to such a data stream. These settings are substantially different as illustrated in Figure 1 (see caption).

We propose a new algorithm for the outlier detection problem in evolving streams, called xStream, to handle this (3)rd, more extreme (hence the name xStream) setting. xStream can also handle settings (1) and (2) above, as (3) is more general as well as more challenging.

A key challenge is the need to maintain not only the outlier detection model but also the data points in memory for the timely processing of the stream. That is, we cannot simply process and discard the points one at a time as they arrive over the stream as

KDD ’18, August 19–23, 2018, London, United Kingdom
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5552-0/18/08…$15.00
https://doi.org/10.1145/3219819.3220107

in setting (2), since those points may "evolve" in two key ways: (*i*) their feature values may change, and/or (*ii*) new features may emerge. Second, both *n* and *d* are unknown; as such, if we were to maintain the data points in memory, space should be dynamically allocated for growing *n* and *d*, both of which could be extremely large. A space cap could cause spillover to disk and as a result "thrashing". Third, a very fast technique is needed to accommodate a likely high-frequency arrival rate of delta-updates per point per feature. Those are specific challenges associated with the (3)rd setting. Other typical challenges include non-stationarity of the data stream, curse of dimensionality, outliers at multiple scales (or granularities) or different subspaces. With xStream, we address all of these challenges above.

While the (3)rd setting for outlier detection has not been considered before, likely because of the challenges associated with evolving data points with potentially high yet unknown dimensionality, the practical cases for this setting are abound. We give three real world examples as follows.

- Data center monitoring: Jobs arriving to a data center over time may evolve, where features like syscall counts, memory usage, #threads, etc. may change or new features like files/repos/URLs accessed or applications run may emerge.
- Customer behavior tracking: Users arriving to a host website over time may evolve, where features like page visit counts, #clicks, etc. may change and new features involving e.g. sequence of visited page types may emerge.
- User monitoring: Active users logging on Twitter/Facebook/etc. over time may evolve, where features like hashtag counts, fraction of posts with URL, URL domain counts, etc. may change and new hashtags or domains may emerge.

In a nutshell, xStream operates on projections of the data points, which it maintains on-the-fly, seamlessly accommodating newly-emerging features. These projections are lower dimensional, fixed size, and preserve the distances between the points well. xStream performs outlier detection via density estimation, through an ensemble of randomized partitions of the data. These partitions are constructed recursively, where the data is split into smaller and smaller bins, which allows us to find outliers at different granularities. Temporal shifts are handled by a window-based approach where bin counts accumulated in the previous window are used to score points in the current window, with windows sliding forward periodically after each current window is full. We give a list of the notable contributions of our work as follows.

- Our most innovative contribution is addressing the outlier detection problem for *evolving* streams for the first time. In such streams, not only new points arrive over time, but feature values may change and new features (columns) may emerge. This is a (3)rd setting where *n* and *d* (the total number of data rows and columns, respectively) are *both unknown*, as compared to (1) the static setting with *n* and *d* both known, and (2) the row-streaming setting with *d* known, *n* unknown.
- We propose a new algorithm called xStream for outlier detection in evolving streams. xStream exhibits a number of key properties: (*i*) it is a **constant memory** approach with fixed-size model and data sketches, processing each stream element in **constant time**; (*ii*) it tackles **high-dimensionality** via detection in subspace projections; (*iii*) it measures outlierness at **multiple scales** or different granularities, which allows identifying both scattered and clustered outliers; and (*iv*) it handles **non-stationarity** via a window-based scheme that requires $O(1)$ update time per window.
- We also show that xStream can easily accommodate settings (1) and (2). In particular, it applies to static **disk-resident** data, requiring only two passes—one for model building and another for scoring outlierness—as well as to row-streaming data, with minor modifications to the implementation.

Through extensive experiments on benchmark datasets, we compare xStream to widely-known state-of-the-art ensemble methods on (1) static and (2) streaming outlier detection. On these settings, xStream is as competitive as existing methods on low-to-medium dimensional datasets, and superior in high-dimensions. To further set xStream apart from the rest, we demonstrate its performance on (3) evolving streams, for which there is no existing competitor.

To facilitate **reproducibility**, we make all source code and data publicly available at https://cmuxstream.github.io/.

2 RELATED WORK

Our work introduces an ensemble-based approach to outlier detection in evolving streams. We cover related work in both areas.

Ensemble Methods: Outlier detection is a well-studied area, especially for static point-cloud data [2, 12]. Following the success of ensemble methods on classification, several ensemble detectors have been developed in the past to leverage "the strength of the many". For example, Lazarevic and Kumar developed feature bagging [22], an ensemble inspired by bagging [8] that employs randomly selected feature subsets to detect outliers in subspaces. Liu et al. [23] propose a method based on random forests [9], which used different subsamples of training data to construct an ensemble of trees to isolate outliers on the basis of the path length from the root to the leaves. Most recently, Aggarwal [3] and Zimek [34] laid out opportunities and challenges in this area, which created a surge of focus on ensemble methods for outlier detection, including the subsampling technique [35] that assembled base detectors with different subsamples of the data points, ensemble of randomized space trees RS-Forest [32], rotated bagging with variable subsampling techniques [4], selective [27] and sequential [28] ensembles, projection-based histogram ensemble LODA [26], and the randomized subspace hashing ensemble RS-Hash [29]. Ensemble techniques enjoy reduced variance and the ability to detect outliers hidden in subspaces. They have been shown to outperform base detectors alone and are considered state-of-the-art for outlier detection.

Streaming Methods: There exist various outlier detectors for data streams [6, 7, 11, 17, 30, 33]. Most related to our work are the ensemble methods that partition the representation space (feature subspaces, projections, etc.) and estimate density based on counts in the partitions [26, 31, 32], which are easily generalized to data streams. The idea is to maintain two separate counts; one for the most recent historical window (called reference) and another for the current window. Then, every time the window slides, reference counts are overwritten by current counts and the latter are set to

Table 1: Comparing xStream with state-of-the-art outlier detection techniques in terms of various properties.

Methods/ Properties	LOF [10]	Feat.Bag. [22]	LOCI [25]	HiCS [21]	iForest [23]	HS-Stream [31]	STORM [6]	LODA [26]	RS-Hash [29]	RS-Forest [32]	xStream
Static	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓
Streaming						✓	✓	✓	✓	✓	✓
Multi-scale			✓								✓
Subspaces		✓		✓	✓	✓		✓	✓	✓	✓
Projections								✓			✓
Evolving feature space											✓
Evolving points/ feature values											✓

zero. Differently, RS-Hash [29] uses continuous counting (i.e., no sliding windows) where points are down-weighted by recency.

While there is much work on time series outlier detection [18], this differs from outlier detection in data streams. Here, the goal is to identify outlying time-series patterns based on temporal dependencies, rather than independent outlying objects.

Remarkably, all of the existing techniques consider the row-stream setting where data *objects arrive one at a time* and assume that *dimensionality is known.* None of them can handle either evolving objects with changing feature values, or newly-emerging features where dimensionality is unknown. Our work is the first to address this more challenging setting and easily generalizes to static as well as row-streaming scenarios as well. Table 1 compares xStream to several popular static and streaming methods in terms of various properties and highlights key differences.

3 OUTLIER DETECTION IN FEATURE-EVOLVING DATA STREAMS

3.1 Problem Definition

We now formalize the problem of outlier detection in feature-evolving data streams. Consider an incoming stream of elements $\mathcal{D} = \{e_t\}_{t=1,2,\ldots}$ where each element e_t is a triple of the form $(id, f, \delta)_t$. Here, the id is a unique integer identifying each data point. f is a feature *name*, represented as a string. δ is a discrete or continuous scalar.

Each such triple is an *update* to the point with identifier id lying in a feature space $\mathcal{F}$ that is *unobserved*. We do not assume anything about the types of features: they may be categorical or numerical. The update is made to a feature with name f and is of magnitude δ. For example: the triple (1000, "sudo", +2) increments the number of times a user with ID 1000 ran the sudo command on a machine, and the triple (3, "rainfall_2018.02", 2.5) records the amount of rainfall observed by a sensor with ID 3 on a specific date.

We define our outlier detection problem formally as follows:

Problem 1 (Outlier Detection in Feature-Evolving Data Streams). *Given a stream* $\mathcal{D} = \{e_t\}_{t=1,2,\ldots}$ *of triples* $e_t = (id, f, \delta)_t$, *compute and maintain an outlier score for each evolving point id such that outliers are scored higher than non-outlier points at any time t.*

Proposed Method: xStream

We now propose xStream to detect outliers in feature-evolving data streams. xStream is an ensemble of *Half-Space Chains* that approximates density efficiently, without needing to know the underlying feature space a-priori. Each chain approximates the density of a point by counting its nearby neighbors at multiple scales. The method builds on the following key components: (1) StreamHash: subspace-selection and dimensionality reduction via sparse random projections for evolving feature spaces; (2) Half-Space Chains: an efficient ensemble to estimate density at multiple scales; and (3) extensions to handle non-stationarity and evolving data points in the stream. We now describe each of these components, followed by the complete algorithm and its complexity in space and time.

3.2 StreamHash

Random projections are an efficient and effective method of reducing data dimensionality while accurately preserving distances between points. Classical random projections [19] involve drawing a set of Gaussian random vectors $\{\boldsymbol{r}_1, \ldots, \boldsymbol{r}_K\} \subset \mathbb{R}^d$ and projecting each point $\boldsymbol{x} \in \mathbb{R}^d$ to a low-dimensional embedding $\boldsymbol{y} \in \mathbb{R}^K$ as: $\boldsymbol{y} = (\boldsymbol{x}^T\boldsymbol{r}_1, \ldots, \boldsymbol{x}^T\boldsymbol{r}_K)$, where K is the number of random projections. This low-dimensional embedding preserves pairwise distances between points with high-accuracy [20].

In high-dimensional data, outliers often lie in low-dimensional subspaces, rendering them difficult to detect with a large number of irrelevant features [5, 36]. This difficulty may be mitigated by looking for outliers in selected subspaces of the data [2]. Hence, we consider a variant of random projections called database-friendly random projections [1], which replaces the Gaussian random vectors with sparse random vectors where only 1/3 of the vector components are non-zero. Under this scheme, each projection ignores a 2/3 fraction of the feature space, essentially selecting a subspace, while still maintaining the ability to preserve pairwise distances. Specifically, the projection values $\boldsymbol{r}_i[j]$ follow the distribution:

$$\boldsymbol{r}_i[j] = \sqrt{\frac{3}{K}} \begin{cases} -1 & \text{with probability } \frac{1}{6} \\ 0 & \text{with probability } \frac{2}{3} \\ +1 & \text{with probability } \frac{1}{6} \end{cases}$$

In a feature-evolving stream, however, the true dimensionality d is unobserved and evolving over time. Hence, it is impossible to draw random vectors $\boldsymbol{r}_1, \ldots, \boldsymbol{r}_K$ of this dimensionality a-priori. To address this issue, we propose StreamHash, a hashing-based implementation of sparse random projections that functions in unobserved and evolving feature spaces. StreamHash is instantiated with K hash-functions $h_1(\cdot), \ldots, h_K(\cdot)$. Each hash function h_i maps a string f (the feature name) to a hash-value, i.e., $h_i : f \to \mathbb{R}$. Given a point $\boldsymbol{x}$ in a fixed feature space $\mathcal{F}$, its random projection via StreamHash is computed as follows:

$$\boldsymbol{y}[i] = \sum_{f_j \in \mathcal{F}} h_i(f_j)\, \boldsymbol{x}[j], \quad i = 1, \ldots, K. \tag{1}$$

Eq. (1) applies to data points arriving with all their features at once. In a feature-evolving stream $\mathcal{D}$ of triples (id, f, δ), updates to the projection of point id are computed as follows:

$$\boldsymbol{y}_{id}[i] = \boldsymbol{y}_{id}[i] + h_i(f)\, \delta, \quad i = 1, \ldots, K. \tag{2}$$

If $\boldsymbol{y}_{id}$ did not exist on the arrival of the first triple, it is initialized with the first such update. Hence, with this projection scheme, we can now compute and maintain low-dimensional projections of the incoming data points in an online fashion, without observing the underlying feature space.

What remains is to specify an implementation of the hash functions h_i's such that the hash function values $h_i(f)$ follow the distribution of projection values $\boldsymbol{r}_i[j]$ in Eq. (1). Let $g_1(\cdot), \ldots, g_K(\cdot)$ be functions from an efficiently computable universal hash family [14, 15], mapping strings to 32 bit integers. The exact number of bits is irrelevant, and is chosen for efficiency based on the machine word size. Let $a_i(f) = g_i(f)/(2^{32} - 1)$, a number between 0 and 1. $h_i(f)$ is defined for K random projections, analogous to Eq. (1) as:

$$h_i[f] = \sqrt{\frac{3}{K}} \begin{cases} -1 & \text{if } a_i(f) \in [0, 1/6) \\ 0 & \text{if } a_i(f) \in [1/6, 5/6) \\ +1 & \text{if } a_i(f) \in [5/6, 1] \end{cases} \tag{3}$$

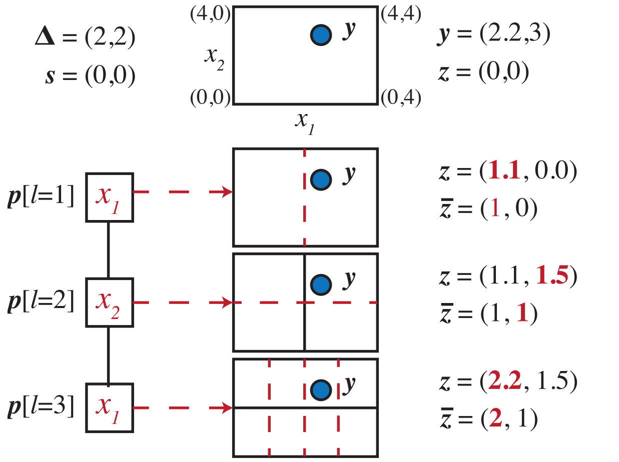
3.3 Half-Space Chains

To detect density-based outliers, it suffices to approximate the density of each point by counting the number of its neighbors lying within some radius r (i.e., at some *scale*). There are two issues with performing neighborhood-counting directly: (1) the success of outlier detection is sensitive to the choice of scale, and (2) in high-dimensional data, the number of neighbors at any scale tends to zero as the dimensionality increases. We resolve both these issues by coupling dimensionality-reduction via StreamHash as described in the previous subsection, with an ensemble that approximates the density by computing neighborhood-counts at multiple scales, which we describe now.

We perform density estimation in the projected K-dimensional space denoted as $\mathcal{P} = \{1, \ldots, K\}$. One approach to approximating density is to construct histograms with equiwidth bins and use the bin-counts. However, as the dimensionality of the binned space increases, the number of bins grows exponentially. This leaves every bin too sparsely populated to reliably approximate the density, as a consequence of the curse of dimensionality. Instead, we propose an ensemble of *Half-Space Chains* of depth D. Each chain randomly selects a single *split-dimension* $p \in \mathcal{P}$ at each *level* $l = 1, \ldots, D$, and recursively splits the space along that dimension into discrete bins. Additionally, if a feature is sampled again at subsequent levels in the chain, it is discretized with a smaller bin width, thus enabling density approximation at multiple scales.

Let us denote by $\boldsymbol{p}$ the vector of split features sampled by a chain, where $\boldsymbol{p}[l] \in \mathcal{P}$ for each level $l = 1, \ldots, D$. The split features are sampled uniformly at random with replacement; this allows dimensions to be split into finer granularities if sampled multiple times. Let $\boldsymbol{\Delta}[p]$ be the initial bin-widths, used to discretize dimensions when sampled for the first time. This is set to approximately half the range of the projected data along each dimension p, and may be estimated by an initial sample.

We approximate the density at a projected point $\boldsymbol{y}$ at level l of the chain by binning the point at that level and retrieving the bin-count. Let the bin-vector (corresponding to a unique bin-id) be $\bar{\boldsymbol{z}} \in \mathbb{Z}^K$, initialized to $\boldsymbol{0}$. Let $p = \boldsymbol{p}[1] \in \mathcal{P}$ be the feature sampled by the chain at level $l = 1$. The bin-vector at $l = 1$ is computed using the initial bin-width as $\bar{\boldsymbol{z}}[p] = \lfloor \boldsymbol{y}[p]/\boldsymbol{\Delta}[p] \rfloor$. Every subsequent time p is sampled in the chain, the bin-width is halved and used to update the bin-vector as above.



$\boldsymbol{\Delta}$ = (2,2)
$\boldsymbol{s}$ = (0,0)
(4,0)
(4,4)
(0,0)
(0,4)
x_2
x_1
$\boldsymbol{y}$
$\boldsymbol{y}$ = (2.2,3)
$\boldsymbol{z}$ = (0,0)
$\boldsymbol{p}[l$=1]
x_1
$\boldsymbol{z}$ = (**1.1**, 0.0)
$\bar{\boldsymbol{z}}$ = (1, 0)
$\boldsymbol{p}[l$=2]
x_2
$\boldsymbol{z}$ = (1.1, **1.5**)
$\bar{\boldsymbol{z}}$ = (1, **1**)
$\boldsymbol{p}[l$=3]
x_1
$\boldsymbol{z}$ = (**2.2**, 1.5)
$\bar{\boldsymbol{z}}$ = (**2**, 1)

Figure 2: Example illustration of a Half-Space Chain (left) of depth $D = 3$ and $\boldsymbol{\Delta}$, $\boldsymbol{s}$ as shown in a $K = 2$ dimensional projected space (top). The dimension selected at each level $\boldsymbol{p}[l{=}1]$, $\boldsymbol{p}[l{=}2]$ and $\boldsymbol{p}[l{=}3]$ is recursively split into bins (middle). Corresponding un-discretized and discretized bin-vectors, respectively $\boldsymbol{z}$ and $\bar{\boldsymbol{z}}$, at each level are shown (right, last computed bin element in bold). See §3.3 for details.

For binning-based density estimation, the placement of the bin boundaries plays a crucial role; which may have an adverse effect on points near the boundaries, potentially causing clusters to split due to lying on both sides. To tackle this issue, we introduce a random *shift* $\boldsymbol{s}[p]$ for each dimension $p \in \mathcal{P}$, such that $\boldsymbol{s}[p] \in \text{Uniform}(0, \boldsymbol{\Delta}[p])$. These shifts reduce the impact of deterministic bin boundaries on clustered points by providing them an opportunity to fall into the same bin. Let $o(p, l)$ be the number of times feature $p \in \mathcal{P}$ has been sampled in the chain until and including level l, where $o(p, l) = 1$ at the first level p is sampled. The bin-vector of $\boldsymbol{y}$ at level l is then computed as,

$$\boldsymbol{z}[p] = \frac{\boldsymbol{y}[p] + \boldsymbol{s}[p]/2^{o(p,l)-1}}{\boldsymbol{\Delta}[p]/2^{o(p,l)-1}}, \quad \forall\, p \in \mathcal{P} \tag{4}$$

$$\bar{\boldsymbol{z}} = \lfloor \boldsymbol{z} \rfloor \tag{5}$$

where $\boldsymbol{z} \in \mathbb{R}^K$ is the *un-discretized bin-vector* of $\boldsymbol{y}$. Fig. 2 illustrates the process of recursively splitting a 2-dimensional projected space by a Half-Space Chain of depth $D = 3$, bin-widths $\boldsymbol{\Delta} = (2, 2)$ without shifts, i.e. $\boldsymbol{s} = (0, 0)$ for simplicity. At levels $l = 1$ and 2, dimensions $\boldsymbol{p}[1] = x_1$ and $\boldsymbol{p}[2] = x_2$ are selected for the first time and split into two bins each of width $\boldsymbol{\Delta}[1] = 2$ and $\boldsymbol{\Delta}[2] = 2$. At level $l = 3$, dimension $\boldsymbol{p}[3] = x_1$ is split again to a finer granularity, into bins of width $\boldsymbol{\Delta}[1]/2 = 1$. Corresponding $\bar{\boldsymbol{z}}$ and $\boldsymbol{z}$ at each level are shown on the right.

Given $\bar{\boldsymbol{z}}$ at any level, we index $\bar{\boldsymbol{z}}$ into a counting data-structure maintained at that level to obtain the bin-count of the point. However, the number of unique bin vectors (esp. at higher granularities or levels) can be large and in fact is theoretically unbounded in a streaming scenario. Hence, we use a *count-min-sketch* $H_l \in \mathcal{H}$ at each level $l = 1, \ldots, D$, which approximates the bin-counts at that level with high accuracy while consuming constant space [13]. Overall, each chain is defined by $C = \{\boldsymbol{p}, \boldsymbol{\Delta}, \boldsymbol{s}, \mathcal{H}\}$ and the collection of M such chains $\mathcal{C} = \{C_1 \ldots, C_M\}$ constitutes xStream's ensemble of Half-Space Chains.

In-place construction of bin-vectors. We may use Eqs. (4) and (5) to compute the bin-vectors at each level $l = 1, \ldots, D$. However, the following recursive relationship enables computing the un-discretized bin-vectors *incrementally*; starting at $l = 1$ and moving downward ($\boldsymbol{z} = \boldsymbol{0}$ initially) using a few arithmetic operations:

$$\begin{aligned} \boldsymbol{z}[p] &= (\boldsymbol{y}[p] + \boldsymbol{s}[p])/\boldsymbol{\Delta}[p] \quad \text{if } o(p,l) = 1 \\ \boldsymbol{z}[p] &= 2\boldsymbol{z}[p] - \boldsymbol{s}[p]/\boldsymbol{\Delta}[p] \quad \text{if } o(p,l) > 1 \end{aligned} \tag{6}$$

where p is the feature sampled at level l, and $o(p, l)$ denotes the number of times feature $p \in \mathcal{P}$ has been sampled as defined earlier.

Updating bin-counts in a Half-Space Chain. On the arrival of a projected point $\boldsymbol{y}$, its bin-vector $\bar{\boldsymbol{z}}$ is constructed at each level $l = 1, \ldots, D$ recursively using Eq. (6) and is indexed into H_l to increment the bin-count $H_l[\bar{\boldsymbol{z}}]$. One may also need to delete points from the chain: we will see in §3.4 that, when points evolve, their old projected points are deleted from the chain and updated ones are added. Deletion proceeds exactly the same as addition, with the bin-count being decremented instead.

Multi-scale Outlier Scoring. At higher levels, the bin-counts tend to be lower. To facilitate comparing outlier scores across levels, we define the outlier score of $\boldsymbol{y}$ (with bin-vector $\bar{\boldsymbol{z}}$) at each level l as the extrapolated bin-count $2^l H_l[\bar{\boldsymbol{z}}]$. This is equal to the expected number of points in each bin if the initial bin-widths $\boldsymbol{\Delta}[p]$ for $p \in \mathcal{P}$ were exactly half the range of p, and the data points were distributed uniformly at random.

We define the multi-scale outlier score of a point $\boldsymbol{y}$ from a chain C as the minimum extrapolated bin-count across all levels, corresponding to the lowest density this point has among all the considered granularities. The overall outlierness score by the ensemble is then the average of the scores across all chains, that is:

$$S(\boldsymbol{y}) = \frac{1}{M} \sum_{C \in \mathcal{C}} S_C(\boldsymbol{y}) = \frac{1}{M} \sum_{C \in \mathcal{C}} \min_l 2^l H_l[\bar{\boldsymbol{z}}] \tag{7}$$

Note that we update the bin-counts and compute outlierness scores simultaneously as we traverse each chain from $l = 1$ downward.

3.4 Handling Non-stationarity and Evolving Points

Streaming random projections (§3.2) coupled with Half-Space Chains (§3.3) can be used to compute outlier scores over any stream of incoming data points in evolving feature space. However, the distribution of points may change as the stream progresses, causing bin-counts constructed in the past to no longer represent the current distribution of the data. Additionally, triples in the data stream may update previously seen points. As such, we next extend xStream to handle non-stationarity and evolving data points.

Handling non-stationarity. We handle non-stationarity by maintaining separate bin-counts for an alternating pair of windows containing ψ points each, termed as *current* and *reference* windows.

Algorithm 1 xStream

```
Input Stream $\mathcal{D}$; Half-Space Chains $\mathcal{C}$; Window size $\psi$
Output Outlier score for each $id$ in $\mathcal{D}$, at any instant
 1: cache = []                                        ▷ Size-$N$ LRU cache
 2: numpoints = 0
 3: for $e_t = (id, f, \delta)_t \in \mathcal{D}$ do
 4:     if $id \notin$ cache then                      ▷ previously unseen $id$, §3.4
 5:         if numpoints = $\psi$ then
 6:             numpoints ← 0
 7:             for $H_l \in \mathcal{H}_C, \forall C \in \mathcal{C}$ do
 8:                 $H_{l.r} \leftarrow H_{l.c}$
 9:                 $H_{l.c} \leftarrow 0$
10:         numpoints ← numpoints +1
11:     else                                          ▷ evolving existing $id$, §3.4
12:         $\boldsymbol{y}_{id} \leftarrow$ Remove $(id, \boldsymbol{y}_{id})$ from cache
13:         Delete $\boldsymbol{y}_{id}$ from the current counters of $\mathcal{C}$
14:     $\boldsymbol{y}_{id} \leftarrow$ Get updated projection for triple $e_t$   ▷ Eq. (2)
15:     Add $(id, \boldsymbol{y}_{id})$ to (head of) cache
16:     Add $\boldsymbol{y}_{id}$ to the current counters of $\mathcal{C}$
17:     $S(\boldsymbol{y}_{id}) \leftarrow$ score with reference counters of $\mathcal{C}$ ▷ Eq. (7)
18:     return $S(\boldsymbol{y}_{id})$
```

These are associated with two separate counters in each entry of a count-min-sketch H_l, denoted by $H_{l.c}$ and $H_{l.r}$ for the current and reference window respectively. As a point $\boldsymbol{y}$ is propagated down the chain, the current window bin-counts $H_{l.c}[\bar{\boldsymbol{z}}]$ are incremented by 1 at each level whereas the reference window counts $H_{l.r}[\bar{\boldsymbol{z}}]$ are used to compute is outlierness score. On the arrival of $(\psi + 1)^{\text{th}}$ new point, the counts in $H_{l.r}$ are replaced with those in $H_{l.c}$, and the counts in $H_{l.c}$ are set to zero to begin processing the next window. The swapping and resetting of counts, i.e., model update is effectively $O(1)$.

Here, ψ is set based on the expected frequency of distribution-changes in the stream; a higher frequency would benefit from lower ψ. However, setting ψ too low reduces the statistical power of the reference window, resulting in outlier scores with a high variance.

Handling evolving data points. Points may *evolve* by receiving updates in the stream to either an existing feature or to a new, previously unseen feature. In either case, Eq. (2) requires the existing projected point $\boldsymbol{y}$ to reside in main memory so as to update it quickly without accessing the disk.

Hence, we maintain in memory a fixed-size cache of N projected points. We follow a Least-Recently-Updated (LRU) eviction protocol for the cache: addition of a new point to a full cache is followed by the eviction of the least recently updated point. The LRU cache is implemented as a linked-list, with additions to the head and evictions from the tail. On receiving an update, the existing projected point is first removed from the cache and from the chains by decrementing its bin-counts at all levels. It is then updated using Eq. (2), moved to the head of the cache and used to increment the corresponding bin-counts in the chains.

Overall xStream **algorithm**, including the mechanisms to handle non-stationarity and evolving data points, is provided in Algorithm 1. The window mechanism is depicted in lines 4–10, and the caching mechanism in lines 11–15. Note that lines 16-17 are performed simultaneously during the same traversal of each chain.

3.5 Time and Space Complexity

Time: Given arriving stream element e_t, we update the projected vector $\boldsymbol{y}$ in $O(K)$. For a given chain in the ensemble, we then construct the bin-vector $\bar{\boldsymbol{z}}$ and index it into the count-min-sketch (CMS) at every level. Here, indexing involves hashing $\bar{\boldsymbol{z}}$ into m different fixed-size CMS hashtables (the larger the m, the better the approximate counts). This takes $O(Km)$ at each level. For a total of M chains each with depth D, time complexity per incoming update becomes ($KmDM$), which is a constant.

Space: xStream maintains (*i*) M half-space chains and (*ii*) N evolving (projected) points in main memory. The CMS hashtables per level constitute the main space requirement for a chain. That is, for m size-L hashtables at D levels, the space for the ensemble structure is $O(mLDM)$. Together with the LRU cache, overall requirement becomes $O(MmLD + NK)$, also a constant.

For e.g., an ensemble with M=100, m=10, L=1000, D=10 plus a cache of N=100, 000 points with K=100 would require 160MB.

Note that our space overhead is linear in depth by design; unlike for e.g., tree structures employed by existing (static/row-streaming) outlier ensembles [23, 31, 32], which grow exponentially by depth.

3.6 xStream for Static and Row-Streaming Data

Next we demonstrate that xStream can easily accommodate settings (1) and (2) as we discussed in the Introduction.

Note that the half-space chains ensemble structure can be constructed *without data*. That is, the split dimension and shift for each level and the fixed-size count-min-sketches can be instantiated apriori. Moreover, K d-dimensional sparse projector vectors ($\boldsymbol{r}_i$'s) can be created and kept in memory.

For disk-resident static setting, xStream then makes two passes over the data. In pass 1, it populates the counters. That is, it loads one point/row $\boldsymbol{x}$ at a time, computes projected vector $\boldsymbol{y}$ and the corresponding bin-ids at each level of each chain using Eq.(6) to increment the count-min-sketch counters. In pass 2, it scores for outlierness. That is, it follows the same steps as in pass 1 but this time uses the populated counts in corresponding count-min-sketch bins to compute outlierness per point as in Eq. (7).

For the row-streaming setting, xStream implementation is almost identical to the disk-resident static setting above, since reading one point at a time from disk is analogous to receiving one point at a time over the stream. The difference is that this time xStream employs the window mechanism where it uses current and reference counters to populate and score points, respectively. Note that in this setting, xStream can still create and cache the K projectors in memory apriori, since dimensionality d is known for row-streams.

4 EVALUATION

4.1 Experiment Settings

We evaluate and compare xStream to various state-of-the-art baselines under three different settings.

Static. This is the offline setting, where both n and d are known and all of the data is available at train time. We use all the data to train detection models and subsequently score the same points under the trained model. Note that all the models used for detection are unsupervised, as such, training does not make ground truth labels available to the models but only used for evaluation.

Table 2: Datasets used for evaluation.

Name	Evolving $\mathcal{F}$?	Evolving points?	n or $\lvert\mathcal{D}\rvert$	d	No. of outliers
gisette	No	No	3850	4970	351
isolet	No	No	4886	617	389
letter	No	No	4586	617	389
madelon	No	No	1430	500	130
cancer	No	No	385	30	28
ionosphere	No	No	242	33	17
telescope	No	No	13283	10	951
indians	No	No	538	8	38
Spam-SMS	Yes	No	5574	8442	747
Spam-URL	Yes	No	2.4M	3.2M	792K
Attk-Flash	Yes	Yes	63.1M	1.1M	2.8M
Attk-Java	Yes	Yes	89.7M	1.1M	29.5M

Datasets. We use the top eight static datasets in Table 2 with varying size and dimensionality. These are standard UCI datasets previously used for outlier detection (see [26] for details).

Baselines. In static setup, we compare xStream to four techniques: *iForest* [23], *HS-Trees* [31], *LODA* [26], and *RS-Hash* [29]. These are all ensemble methods that seek outliers in subspaces, and have been shown to outperform numerous standard detectors [4] and hence are considered the state-of-the-art.

We use recommended sample size ψ = 256 and height limit `hlim`= 15 for *iForest*, max-depth 15 for *HS-Trees*, sparsity factor $1/\sqrt{d}$ for *LODA* and let it select the best width for its histograms, the recommended sample size 1000 for *RS-Hash*, and k = 100 and ℓ = 15 for our xStream. For all methods, we set the number of ensemble components (trees, histograms, chains) to 100.

Row-stream. The second setting is where data points with *known* dimensionality arrive *one-at-a-time* over the stream. Each arriving point is scored under the detection model and discarded.

Datasets. We use the Spam-SMS and Spam-URL datasets listed in the middle part of Table 2, containing ground truth spam SMS messages and emails containing spam URLs, respectively. Each SMS or URL document arrives one at a time over the stream. As such, these datasets well-align with the row-streaming setting—except for one caveat, which is their feature representations. We use the word and shingle counts as features for both type of documents, however, the vocabulary size in reality is unknown apriori (esp. for SMS, where the vocabulary is highly variable). That is, the feature domain $\mathcal{F}$ is in fact evolving. Nevertheless, we give the advantage of known-d to existing methods, by identifying all unique features in the entire corpus apriori and constructing feature vectors accordingly for the incoming documents.

Baselines. We compare to the streaming versions of the baseline techniques from the previous part, namely *HS-Stream* and *LODA* (*iForest* does not have a stream version). We use similar hyperparameter settings as before. In addition, we use two sliding windows (reference and current) for *HS-Stream*, *LODA*, and xStream with various window sizes. *RS-Hash* uses a continuous counting approach with (recommended) decay rate 0.015.

Note that our work addresses a much more challenging outlier detection problem than the offline and row-streaming (w/ known-*d*) scenarios above. We provide comparative results solely for completeness and to show xStream's competitiveness in these settings.

Evolving Stream. This is the extreme setting we consider in this work, in which *updates* arrive over time as a stream $\mathcal{D}$. These may include updates to existing features' values as well as newly-emerging features of new or previous data points.

Datasets. We use the Attk-Flash and Attk-Java datasets listed in the last section of Table 2, also used in our prior work [24]. They contain traces from processes executing simultaneously on a machine, respectively from malicious processes exploiting a Flash or Java vulnerability. Each dataset also contains traces each from 5 non-attack scenarios corresponding to casual browsing activities: YouTube, GMail VGame (a Flash game), Download (downloading a file) and CNN (a news website). Here, processes do not arrive one at a time as in setting (2), rather, they run in parallel. Moreover, they *evolve*; executing various system calls (e.g., file read/port write/etc.) over time. We represent the sequence of calls by a process with shingle (subsequence) counts, where new shingles emerge in time. Moreover, such shingle updates arrive to any and all processes running on the host in an interleaved fashion, requiring the detector to maintain the representation of "live" processes and update their outlier scores over time.

Baselines. In this new setting, there is no existing competitor. xStream is the *first* technique (to the best of the authors' knowledge) for outlier detection in evolving data streams—handling evolving feature space and evolving feature values.

4.2 Static Experiments

Table 3 reports the performance of competing methods as measured by average precision (area under the precision-recall curves) on static datasets. The results are averaged across 10 independent runs, since all methods are non-deterministic.

We statistically compare the average ranks of the algorithms (reported in Table 3) using the nonparametric Friedman test [16]. With $p = 0.1107$, we could not reject the null hypothesis at a 5% confidence-level that all methods are equally performant, suggesting xStream is as competitive as the state-of-the-art in this basic setting. Next, we perform the Nemenyi post-hoc test to compare the methods in pairs while accounting for multiple testing. The test reports significance if the average ranks of two methods differ by a "critical distance" (CD): for 5 methods on 8 datasets at a significance level $\alpha = 0.05$, the CD = 2.1567. Since the gap between the lowest (by xStream) and highest (by *iForest*) average ranks is already less than the CD, we again conclude that there is no significant difference in performance between any pair of methods.

xStream is designed for high-dimensional data with many newly-emerging features. With many dimensions, we expect most features to be irrelevant for outlier detection. Put differently, outliers tend not to stand out in all possible dimensions. To showcase such a scenario, we append new columns with Gaussian noise to the four lowest dimensional static datasets, thereby increasing the dimensionality with noisy features. Specifically we append 100%, 1000%,

Table 3: Mean Average Precision on static datasets. Mean and standard deviation are reported over 10 runs.

Dataset	*iForest*	*HS-Trees*	*RS-Hash*	*LODA*	xStream
cancer	0.617 ± 0.021	0.646 ± 0.033	0.619 ± 0.030	0.826 ± 0.013	0.845 ± 0.008
ionosphere	0.705 ± 0.006	0.706 ± 0.007	0.764 ± 0.032	0.642 ± 0.067	0.848 ± 0.018
telescope	0.367 ± 0.008	0.392 ± 0.012	0.391 ± 0.012	0.322 ± 0.007	0.344 ± 0.009
indians	0.142 ± 0.003	0.146 ± 0.002	0.156 ± 0.007	0.177 ± 0.008	0.216 ± 0.010
gisette	0.078 ± 0.002	0.080 ± 0.002	0.084 ± 0.007	0.087 ± 0.003	0.090 ± 0.003
isolet	0.099 ± 0.003	0.097 ± 0.005	0.108 ± 0.004	0.089 ± 0.004	0.112 ± 0.006
letter	0.093 ± 0.001	0.092 ± 0.002	0.104 ± 0.004	0.094 ± 0.006	0.122 ± 0.005
madelon	0.110 ± 0.003	0.101 ± 0.013	0.092 ± 0.005	0.101 ± 0.010	0.097 ± 0.004
Avg Rank	3.75	3.3125	2.875	3.3125	1.75

Table 4: Mean Average Precision on static perturbed datasets, reported over 10 runs. Numbers in brackets indicate the number of Gaussian noise columns as a % of the original dimensionality.

Dataset	*iForest*	*HS-Trees*	*RS-Hash*	*LODA*	xStream
cancer (100)	0.599 ± 0.031	0.605 ± 0.031	0.646 ± 0.032	0.811 ± 0.012	0.825 ± 0.012
cancer (1K)	0.406 ± 0.088	0.201 ± 0.024	0.425 ± 0.112	0.722 ± 0.056	0.813 ± 0.022
cancer (2K)	0.306 ± 0.044	0.229 ± 0.029	0.337 ± 0.077	0.633 ± 0.092	0.822 ± 0.021
cancer (5K)	0.120 ± 0.040	0.158 ± 0.018	0.153 ± 0.070	0.336 ± 0.141	0.796 ± 0.028
i.sphere (100)	0.651 ± 0.049	0.568 ± 0.026	0.622 ± 0.038	0.560 ± 0.056	0.848 ± 0.011
i.sphere (1K)	0.302 ± 0.072	0.231 ± 0.006	0.258 ± 0.070	0.589 ± 0.073	0.819 ± 0.019
i.sphere (2K)	0.211 ± 0.105	0.085 ± 0.007	0.233 ± 0.100	0.561 ± 0.092	0.791 ± 0.026
i.sphere (5K)	0.112 ± 0.035	0.150 ± 0.017	0.135 ± 0.062	0.494 ± 0.072	0.685 ± 0.065
t.scope (100)	0.311 ± 0.012	0.260 ± 0.006	0.326 ± 0.015	0.322 ± 0.006	0.340 ± 0.008
t.scope (1K)	0.156 ± 0.011	0.102 ± 0.004	0.164 ± 0.019	0.303 ± 0.010	0.311 ± 0.006
t.scope (2K)	0.108 ± 0.010	0.098 ± 0.014	0.112 ± 0.019	0.296 ± 0.016	0.284 ± 0.005
t.scope (5K)	0.084 ± 0.005	0.079 ± 0.001	0.087 ± 0.011	0.248 ± 0.017	0.271 ± 0.005
indians (100)	0.123 ± 0.007	0.093 ± 0.003	0.128 ± 0.009	0.171 ± 0.008	0.196 ± 0.015
indians (1K)	0.086 ± 0.014	0.096 ± 0.009	0.087 ± 0.011	0.153 ± 0.028	0.178 ± 0.006
indians (2K)	0.087 ± 0.013	0.076 ± 0.003	0.085 ± 0.008	0.139 ± 0.028	0.151 ± 0.013
indians (5K)	0.073 ± 0.007	0.075 ± 0.009	0.083 ± 0.018	0.126 ± 0.028	0.152 ± 0.020
Avg Rank	4.0625	4.4375	3.25	2.1875	1.0625

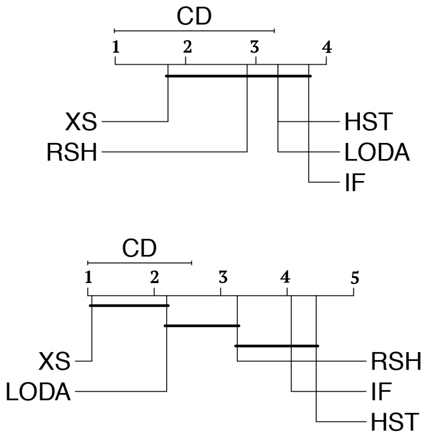


Figure 3: Comparison of detectors in terms of average rank with the Nemenyi test on (top) original and (bottom) noisy static datasets. Groups of methods that are not significantly different ($p = 0.05$) are connected. CD is the critical distance required to reject equivalence.

Table 5: Mean average precision (MAP) and overall average precision (OAP) on SPAM-SMS. Window-size set as a percentage of the rows in the entire dataset. xSTREAM-1K denotes our method using $M = 1000$ half-space chains (instead of 100).

Window	***HS-Stream***		***LODA***		***RS-Hash***		xSTREAM		xSTREAM-**1K**	
size ψ	*MAP*	*OAP*	*MAP*	*OAP*	*MAP*	*OAP*	*MAP*	*OAP*	*MAP*	*OAP*
1%	0.480 ± 0.178	0.416	0.090 ± 0.028	0.076	0.291 ± 0.129	0.171	0.505 ± 0.138	0.422	0.522 ± 0.153	0.430
5%	0.492 ± 0.179	0.416	0.082 ± 0.014	0.077	0.216 ± 0.034	0.195	0.455 ± 0.135	0.406	0.493 ± 0.134	0.415
10%	0.430 ± 0.024	0.419	0.081 ± 0.010	0.080	0.174 ± 0.017	0.164	0.444 ± 0.037	0.433	0.448 ± 0.037	0.436
25%	0.363 ± 0.024	0.359	0.080 ± 0.001	0.080	0.203 ± 0.014	0.201	0.409 ± 0.009	0.404	0.435 ± 0.013	0.429

Table 6: Mean average precision (MAP) and overall average precision (OAP) on SPAM-URL. Window size set as days of incoming URLs, for a total of ≈20,000 URLs per-day.

Window	***HS-Stream***		***LODA***		***RS-Hash***		xSTREAM		xSTREAM-**1K**	
size ψ	*MAP*	*OAP*	*MAP*	*OAP*	*MAP*	*OAP*	*MAP*	*OAP*	*MAP*	*OAP*
1 day	0.331 ± 0.055	0.330	0.329 ± 0.059	0.331	0.358 ± 0.061	0.357	0.436 ± 0.083	0.437	0.451 ± 0.106	0.452
3 days	0.339 ± 0.058	0.329	0.307 ± 0.055	0.328	0.357 ± 0.046	0.356	0.478 ± 0.078	0.479	0.508 ± 0.064	0.509
5 days	0.336 ± 0.059	0.329	0.321 ± 0.044	0.328	0.357 ± 0.038	0.356	0.472 ± 0.050	0.472	0.493 ± 0.055	0.496
7 days[1]	—	—	0.303 ± 0.040	0.321	0.355 ± 0.036	0.356	0.497 ± 0.053	0.502	0.533 ± 0.049	0.530

[1] *HS-Stream* exceeds the available memory on a 1 TB machine.

2000% and 5000% noisy features of the original dataset dimensionality, with noise $\sim N(.1\mu, .1\sigma)$ where μ and σ are the mean and tandard deviation of all features in the original dataset.

Table 4 lists the mean average precision along with average ranks of each method. This time, we can reject the null with $p = 5.565 \times 10^{-10}$ using the Friedman test. We proceed with the Nemenyi post-hoc test to find out which detectors actually differ significantly. For 5 methods on 16 (perturbed) datasets at a significance level $\alpha = 0.05$, the CD = 1.524. As shown in Table 4, we find a significant difference between xSTREAM and all baselines except *LODA*. As expected, projection-based methods are superior on high-dimensional noisy datasets, while the performance drops gradually for *iForest*, *HS-Trees*, and *RS-Hash* as the number of noisy features increases.

We summarize our findings through a graphical presentation of results in Figure 3. On average, xSTREAM is as competitive with no significant difference to existing baselines in the static setting. This is not a coincidence: intuitively, they all score outliers via density estimation, albeit in different ways. xSTREAM and projection-based *LODA* are advantageous in high dimensions with many irrelevant features, as the (small set of) features carrying signal are less likely to be selected by other methods that work with the original features. Non-projection methods also face the "curse of dimensionality", where the amount of data needed to obtain a statistically reliable density estimate grows exponentially with the dimension.

4.3 Row-Stream Experiments

We report results of all competing methods on SPAM-SMS and SPAM-URL in Table 5 and Table 6 respectively. We evaluate the methods with varying window sizes (ψ). We instantiate the window-based methods xSTREAM, *HS-Stream* and *LODA* using the first window of ψ points. For *RS-Hash* we use an initial sample size of 1000, as recommended in [29]. For SPAM-SMS, the data does not provide an explicit time granularity (like minutes/days); hence, we experiment with ψ = 1, 5, 10, and 25% of the rows. For SPAM-URL, the rows are grouped into collections of all the ham and spam URLs received daily for a total of 120 days. As such, we specify the window size in terms of the number of days; specifically, ψ = 1, 3, 5, and 7 days.

All methods assign an outlier score to each row as soon as it arrives, which remains unchanged henceforth. Using these scores, we compute the average precision for each window containing ψ rows, and report the mean average precision (MAP) over all such windows. Additionally, we report the average precision for all the rows observed at the end of the row stream, denoted as OAP; the overall average precision. Note that the first ψ rows are not assigned scores, since those are used to instantiate the models.

We observe that xSTREAM outperforms *LODA* and *RS-Hash*, and performs on par with *HS-Stream* on SPAM-SMS containing a total of 5574 rows in the stream. On SPAM-URL with 2.4 *million* rows (≈20,000 daily), on the other hand, xSTREAM outperforms *HS-Stream*, *LODA* and *RS-Hash* on average as well as in terms of the end-of-stream performance—where *HS-Trees* ran out of memory for ψ = 7 days on a server with 1TB RAM.

HS-Stream's decayed performance on SPAM-URL containing 3.2 *million* features can be attributed to the dimensionality. While *HS-Stream* operates directly on the high dimensional rows, xSTREAM operates on distance-preserving low-dimensional projections and, as such, is less prone to curse of dimensionality. This advantage is further magnified on datasets where a large number of dimensions are irrelevant for outlier detection (as was demonstrated in §4.2). Though *LODA* also projects the data, its poor performance can be attributed to two factors. First, it uses only 1-dimensional projections, which may not be enough to preserve structure for high-dimensional datasets like SPAM-URL. Second, it relies on discretizing the data using an optimized yet single bin-width. In contrast, xSTREAM employs few but multiple projections, and uses a spectrum of bin-widths to capture outliers at different granularities.

In all experiments, we use 100 ensemble components (trees/histograms/chains) for each method. We also use the same maximum depth for xStream and *HS-Stream*. Note that *HS-Stream* requires space exponential in depth (as it builds balanced binary trees) in contrast to space linear in depth required by xStream. In Table 5 and Table 6, we also present the MAP and OAP for xStream using 1000 chains (thus, increasing the diversity and expressiveness of the ensemble), demonstrating improved performance while still consuming exponentially less space than *HS-Stream*: $O(100 \times 2^{15})$ for *HS-Stream* vs. $O(1000 \times 15)$ for xStream.

Note that the performance of all methods varies with the window-size ψ. This is due to the temporal variation in the distribution of rows over different windows, resulting in varying models being constructed at the start of each window, which are used later for scoring. Hence, no consistent trend is to be expected (nor is observed) as ψ is varied. We experiment with and present results using different window-sizes in order to demonstrate that our conclusions are not biased by the choice of a specific window-size

4.4 Evolving Stream Experiments

We now evaluate xStream on a feature-evolving stream, to detect attack processes executing simultaneously with non-attack processes. For each non-attack scenario, we concatenate a random sample of 25% of the processes from each attack and shuffle the updates, such that the streams interleave but the order of updates in each trace is maintained. xStream is configured with $M = 100, K = 100, D = 15, \psi = 25$ and $m = 8, L = 2^{10}$ for the count-min-sketches.

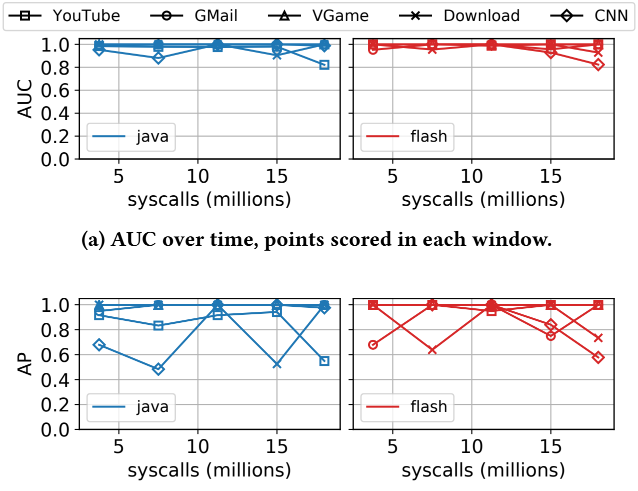
We compute the AP and AUC of the points scored in each window and report them at different instants of the stream in Fig 4 (a)-(b). The MAP over all windows is reported in Fig 4 (c). We observe that the AUC is near-ideal over all windows. The AP fluctuates across windows more than the AUC, but remains ~0.75+ for the majority of datasets and windows. The AP on CNN fluctuates over windows for both Attk-Java and Attk-Flash. The CNN website contains a wide variety of Java and Flash-based rich-media content; this makes the CNN scenario relatively more difficult to distinguish as an outlier using the extracted shingles as features.

To make the setting more realistic, we also combine all the non-attack processes to construct the "All" scenario and again evaluate the performance of xStream in detecting a 25% random sample of each attack. The MAP is noted in the last row of Fig 4 (c). While performance reduces slightly, xStream still remains a competitive outlier detector in this setting.

We analyze the effect of the parameters in the "All" scenario and plot the variation in MAP with each parameter in Fig. 5 (a)-(d).

Increasing M and K each improve the diversity of the ensemble by increasing the number of chains, and increasing the diversity of features sampled in each chain, respectively. The resulting increase in MAP in either case is confirmed in 5 (a)-(b).

As the maximum depth D is increased, xStream is able to detect outliers at finer granularities. However, the gains from increasing D diminish after a point, when the granularity is too fine for the given data. Hence, it is sufficient to use small D in most cases, though a larger D has no negative impact. This is confirmed in Fig. 5(c).



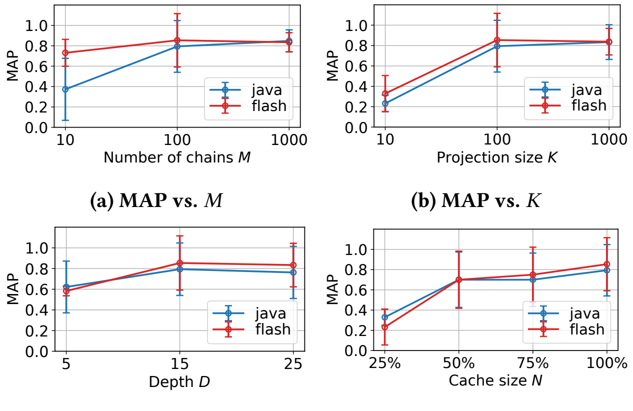
(a) AUC over time, points scored in each window.

(b) AP over time, points scored in each window.

Normal Scenario	Attk-Java	Attk-Flash
YouTube	0.832 ± 0.146	0.990 ± 0.020
GMail	0.990 ± 0.020	0.886 ± 0.142
VGame	0.999 ± 0.001	0.999 ± 0.001
Download	0.905 ± 0.190	0.875 ± 0.156
CNN	0.828 ± 0.211	0.854 ± 0.173
All	0.794 ± 0.254	0.854 ± 0.262

(c) Mean average precision over all windows.

Figure 4: Performance of xStream on feature-evolving streams. $K = 100$, $M = 100$, $D = 15$, $\psi = 25$, $N = 100\%$.



(a) MAP vs. M

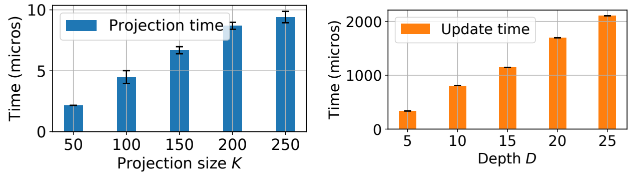
(b) MAP vs. K

(c) MAP vs. D

(d) MAP vs. N

Figure 5: Effect of parameters on xStream's performance. Unless specified, $M = 100$, $K = 100$, $D = 15$, $\psi = 25$, $N = 100\%$.

Finally, the MAP improves as the size of the cache N (in terms of the % of total data points) is increased, as expected and observed in Fig. 5 (d). However, the performance remains competitive even with a cache containing just 50% of the original points.



(a) Projection time vs. K

(b) Update time vs. D

Figure 6: Time for xStream to (a) project a triple with a 32-byte feature name, and (b) update $M = 100$ chains, for different values of sketch size K and maximum depth D. All times are in microseconds.

We also record the time taken for the main operations performed on each incoming update. To facilitate evaluation, we use string feature names that were padded to 32 bytes. Fig. 6 shows the the projection time (Fig. 6(a)) and the time to update all the $M = 100$ chains (Fig. 6(b)) as the projection size K and maximum depth D are varied. The overall time grows linearly with both parameters, as expected. The absolute time is of the order of milliseconds; thus, xStream is capable of processing 1000s of updates per-second.

5 CONCLUSION

We have formulated the problem of detecting outliers in feature-evolving data streams, wherein both the data points and feature space may evolve over time. This setting is novel in the outlier-detection literature with numerous applications, while being inherently more challenging than the static and row-stream settings.

We proposed xStream to solve the problem of outlier-detection in feature-evolving data streams. xStream addresses a number of key challenges in this new setting. To handle large and unobserved dimensionality, it employs a streaming random projection scheme to embed points in a low-dimensional space on-the-fly, while preserving pairwise distances. xStream then detects outliers in this projected space via an ensemble of half-space chains. Each half-space chain of the ensemble enables computing outlier scores for each point via randomized partitions (within subspaces) of the projected input space. Outlier scores are computed at multiple scales, thus facilitating the detection of both clustered and scattered outliers. Further, a window-based mechanism is implemented that handles non-stationarity in the stream. Overall, xStream functions in constant time and constant space for each update, which can be controlled via user-configurable parameters.

Via extensive experiments, we showed that xStream (1) performs on par with the state-of-the art on static data and row-streams, with few changes required, (2) outperforms the state-of-the-art on high-dimensional data streams, and (3) achieves desirable performance on feature-evolving streams while processing over 1000 updates per-second.

ACKNOWLEDGMENTS

This research is sponsored by NSF CAREER 1452425, IIS 1408287, and an Adobe University Marketing Research Award. Any conclusions in this material are of the authors and do not necessarily reflect the views, expressed or implied, of the funding parties.

REFERENCES

[1] Dimitris Achlioptas. 2003. Database-friendly Random Projections: Johnson-Lindenstrauss with Binary Coins. *J. Comput. System Sci.* (2003).

[2] Charu C. Aggarwal. 2013. *Outlier Analysis.* Springer.

[3] Charu C. Aggarwal. 2013. Outlier ensembles. *SIGKDD Explorations* (2013).

[4] Charu C. Aggarwal and Saket Sathe. 2015. Theoretical Foundations and Algorithms for Outlier Ensembles. *SIGKDD Explorations* (2015).

[5] Charu C. Aggarwal and Philip S Yu. 2001. Outlier detection for high dimensional data. *SIGMOD Record* (2001).

[6] Fabrizio Angiulli and Fabio Fassetti. 2007. Detecting distance-based outliers in streams of data. In *CIKM*.

[7] Ira Assent, Philipp Kranen, Corinna Baldauf, and Thomas Seidl. 2012. AnyOut: Anytime Outlier Detection on Streaming Data. In *DASFAA*.

[8] Leo Breiman. 1996. Bagging Predictors. *Machine Learning* (1996).

[9] Leo Breiman. 2001. Random Forests. *Machine Learning* (2001).

[10] Markus M Breunig, Hans-Peter Kriegel, Raymond T Ng, and Jörg Sander. 2000. LOF: identifying density-based local outliers. In *SIGMOD*.

[11] Hui Cao, Yongluan Zhou, Lidan Shou, and Gang Chen. 2010. Attribute Outlier Detection over Data Streams. In *DASFAA*.

[12] Varun Chandola, Arindam Banerjee, and Vipin Kumar. 2009. Anomaly Detection: A Survey. *Comput. Surveys* (2009).

[13] Graham Cormode and S Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms* (2005).

[14] Søren Dahlgaard, Mathias Knudsen, and Mikkel Thorup. 2017. Practical Hash Functions for Similarity Estimation and Dimensionality Reduction. In *NIPS*.

[15] Søren Dahlgaard, Mathias Bæk Tejs Knudsen, Eva Rotenberg, and Mikkel Thorup. 2015. Hashing for statistics over k-partitions. In *FOCS*.

[16] Janez Demšar. 2006. Statistical Comparisons of Classifiers over Multiple Data Sets. *Journal of Machine Learning Research* (2006).

[17] Sudipto Guha, Nina Mishra, Gourav Roy, and Okke Schrijvers. 2016. Robust Random Cut Forest Based Anomaly Detection on Streams. In *ICML*.

[18] Manish Gupta, Jing Gao, Charu C. Aggarwal, and Jiawei Han. 2014. Outlier Detection for Temporal Data: A Survey. *Transactions on Knowledge and Data Engineering (TKDE)* (2014).

[19] Piotr Indyk and Rajeev Motwani. 1998. Approximate nearest neighbors: Towards removing the curse of Dimensionality. In *STOC*.

[20] William B Johnson and Joram Lindenstrauss. 1984. Extensions of Lipschitz mappings into a Hilbert space. *Contemp. Math.* (1984).

[21] Fabian Keller, Emmanuel Müller, and Klemens Böhm. 2012. HiCS: High Contrast Subspaces for Density-Based Outlier Ranking. In *ICDE*.

[22] Aleksandar Lazarevic and Vipin Kumar. 2005. Feature bagging for outlier detection. In *KDD*.

[23] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. 2008. Isolation Forest. In *ICDM*.

[24] Emaad Manzoor, Sadegh M. Milajerdi, and Leman Akoglu. 2016. Fast Memory-efficient Anomaly Detection in Streaming Heterogeneous Graphs. In *KDD*.

[25] Spiros Papadimitriou, Hiroyuki Kitagawa, Phillip B. Gibbons, and Christos Faloutsos. 2003. Fast Outlier Detection Using the Local Correlation Integral. In *ICDE*.

[26] Tomás Pevný. 2016. Loda: Lightweight on-line detector of anomalies. *Machine Learning* (2016).

[27] Shebuti Rayana and Leman Akoglu. 2016. Less is More: Building Selective Anomaly Ensembles. *Transactions on Knowledge Discovery from Data (TKDD)* (2016).

[28] Shebuti Rayana, Wen Zhong, and Leman Akoglu. 2016. Sequential Ensemble Learning for Outlier Detection: A Bias-Variance Perspective. In *ICDM*.

[29] Saket Sathe and Charu C. Aggarwal. 2016. Subspace Outlier Detection in Linear Time with Randomized Hashing. In *ICDM*.

[30] Markus Schneider, Wolfgang Ertel, and Fabio Ramos. 2016. Expected similarity estimation for large-scale batch and streaming anomaly detection. *Machine Learning* (2016).

[31] Swee Chuan Tan, Kai Ming Ting, and Fei Tony Liu. 2011. Fast Anomaly Detection for Streaming Data. In *IJCAI*.

[32] Ke Wu, Kun Zhang, Wei Fan, Andrea Edwards, and Philip S Yu. 2014. RS-Forest: A Rapid Density Estimator for Streaming Anomaly Detection. In *ICDM*.

[33] Ji Zhang, Qigang Gao, and Hai H Wang. 2008. SPOT: A System for Detecting Projected Outliers From High-dimensional Data Streams. In *ICDE*.

[34] Arthur Zimek, Ricardo J.G.B. Campello, and Jörg Sander. 2013. Ensembles for Unsupervised Outlier Detection: Challenges and Research Questions. *SIGKDD Explorations* (2013).

[35] Arthur Zimek, Matthew Gaudet, Ricardo Campello, and Jörg Sander. 2013. Subsampling for efficient and effective unsupervised outlier detection ensembles. In *KDD*.

[36] Arthur Zimek, Erich Schubert, and Hans-Peter Kriegel. 2012. A survey on unsupervised outlier detection in high-dimensional numerical data. *Statistical Analysis and Data Mining: The ASA Data Science Journal* (2012).