SPARx: Distributed Outlier Detection at Scale

Sean Zhang
Carnegie Mellon University
xiaoronz@alumni.cmu.edu

Varun Ursekar
Carnegie Mellon University
vursekar@andrew.cmu.edu

Leman Akoglu
Carnegie Mellon University
lakoglu@andrew.cmu.edu

ABSTRACT

There is no shortage of outlier detection (OD) algorithms in the literature, yet a vast body of them are designed for a single machine. With the increasing reality of already cloud-resident datasets comes the need for distributed OD techniques. This area, however, is not only understudied but also short of public-domain implementations for practical use. This paper aims to fill this gap: We design SPARx, a data-parallel OD algorithm suitable for shared-nothing infrastructures, which we specifically implement in Apache Spark. Through extensive experiments on three real-world datasets, with several billions of points and millions of features, we show that existing open-source solutions fail to scale up; either by large number of points or high dimensionality, whereas SPARx yields scalable and effective performance. To facilitate practical use of OD on modern-scale datasets, we open-source SPARx under the Apache license.[1]

CCS CONCEPTS

• **Computing methodologies** → **Anomaly detection**; **Distributed algorithms**; *MapReduce algorithms.*

KEYWORDS

distributed outlier detection; data-parallel algorithms; Apache Spark

ACM Reference Format:
Sean Zhang, Varun Ursekar, and Leman Akoglu. 2022. SPARx: Distributed Outlier Detection at Scale. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '22), August 14–18, 2022, Washington, DC, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3534678.3539076

1 INTRODUCTION

Motivation. Outlier detection (OD) finds applications in many domains, such as finance [3], manufacturing [27], surveillance [25] and environmental monitoring [31], to name a few. OD is typically used in data cleaning to filter out noise/errors/etc. before fitting a model that may be sensitive to the presence of outliers in the training data. In other settings, outliers are rather the "signal", where OD is employed to identify adversarial or abnormal occurrences, such as network attacks or about-to-fail manufacturing parts.

With the advent of technology, it is typical of applications to generate or collect massive datasets. Often these are already resident in modern distributed infrastructures or cloud services (e.g. Amazon AWS, Microsoft Azure, Google Cloud, etc.), which renders OD algorithms designed for a single machine inapplicable. These datasets may also be ever-growing, with new data being generated in a daily fashion or much faster; such as sensors monitoring data (including social sensors), transactions data, computer network logs data, among others. This trend puts OD algorithms applicable to distributed large-scale datasets in great demand, which is likely to grow further in coming years.

Prior Work. While outlier detection has an extensive literature [2], *distributed* OD is a considerably understudied area. To our knowledge, there exists only a few published work on OD for cloud-resident data on shared-nothing infrastructures. Among those, only a handful of them provide public-domain implementations: DDLOF [32], a distributed implementation of the popular LOF [8] OD algorithm; SPIF [28] is a Spark-based design of the popular (ensemble) algorithm Isolation Forest (IF) [16]; and most recently, DBSCOUT [9] that builds on the ideas from the popular clustering algorithm DBSCAN [13]. DDLOF is based on Hadoop [6], which is 10-100× slower than the in-memory computing platform Apache Spark [34]. On the other hand, the Spark-based SPIF only employs *model-parallelism* (training each ensemble component on a separate compute node); and because it does not leverage data-parallelism, it scales poorly to large datasets with numerous points. DBSCOUT is also built on Spark, however it scales extremely poorly with increasing dimensionality d, and has been tested on 2- and 3-d data only. Moreover, it is a distance-based algorithm that may not work well on data with varying-density support, inherits two sensitive hyperparameters from DBSCAN, and provides only a binary output (inlier/outlier) based on a strict outlier definition. (See Sec. 5 for detailed related work, and Sec. 4 for comparison experiments.)

Besides *i*) scaling-out to *distributed* datasets, there are several other desired characteristics of an OD algorithm for practical usability, including: *ii*) *linear* time and space complexity, *iii*) *robustness* to hyperparameter choices (so that practically easy to use in unsupervised settings), *iv*) carefully handling *high dimensionality*, and *v*) admitting data with *mixed-type features* (both categorical and numerical). Today, no existing OD algorithm in the literature satisfies all these desired properties.

Present Work. Through this work, we set out to make OD a greater contributor to the real-world use cases at large, and introduce a new OD algorithm called SPARx, *exhibiting all the aforementioned practical properties i)–v).* Specifically, we capitalize on the xSTREAM algorithm [20] which is originally designed for a single machine, and transform it to a MapReduce [11] based distributed algorithm. Our implementation is based on the Python API of Apache Spark (hence the name, SPARx) that is suitable for a shared-nothing distributed infrastructure.

SPARx readily inherits all the desirable properties of xSTREAM, while also scaling-out to massive datasets on cloud platforms. In fact, and apart from the extensive experiments in the original paper [20], xSTREAM has recently been externally validated to outperform

KDD '22, August 14–18, 2022, Washington, DC, USA

ACM ISBN 978-1-4503-9385-0/22/08…$15.00
https://doi.org/10.1145/3534678.3539076

a long list of (9) baselines at router-level anomaly detection tasks on 64 different real-world datasets from Huawei Inc. [22]. The algorithms are run using various hyperparameter (HP) configurations, and compared with respect to both best hyperparametrization (optimistic and unrealistic in unsupervised settings) as well as practical (just one) hyperparametrization (realistic yet conservative) as only one HP configuration (as recommended by the original authors) is used for all datasets. xStream performance is outstanding based on their extensive evaluation, quoting: "Remarkably, xStream stands out for being close to the Ideal Ensemble Upper Bound [i.e. best model among all algorithms and HPs]", and "xStream [...] able to provide robust and good performance even in practical settings."

We summarize the main contributions of this work as follows.

- **Distributed OD for Cloud-resident Data**: We present Sparx, a *data-parallel* outlier detection (OD) algorithm for distributed data that handles large number of input points as well as high dimensionality. Sparx is a linear time and space complexity algorithm that can scale-out to datasets that are already cloud-resident. For wide-spread usability, we provide a public-domain implementation of Sparx in Apache Spark.[1]
- **List of Desired Properties**: Sparx is effectively a distributed extension of the xStream algorithm which exhibits many desirable properties; including robustness to hyperparameters, handling high dimensional feature space, admitting mixed-type data, among others. Arguably, all of those combined with scalability to cloud-resident datasets makes Sparx one of the most practically useful, open-source solutions to OD.
- **Scalability and Effectiveness**: We evaluate Sparx against the state-of-the-art baselines DBSCOUT [9] (DBSCAN-centered OD in Spark) and SPIF [28] (Spark-based Isolation Forest). As we show through extensive experiments on datasets with number of points and dimensionality up to several billions and millions, Sparx outperforms SOTA baselines in terms of detection accuracy, running time and memory usage. SPIF fails to scale up to large number of points (limited to model-parallelism), whereas DBSCOUT scales poorly with high dimensionality (does not scale up beyond 10 dimensions).

2 PRELIMINARIES & BACKGROUND

Consider a distributed point-cloud database, originally containing n points with d dimensions; $\mathcal{P} = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$, where $\mathbf{x}_i$'s can be *mixed-type*, i.e. a subset of features being real-valued and the rest being categorical with arbitrary domains. Let $\mathcal{F}$ denote the set of original features where $|\mathcal{F}| = d$.

We consider a general deployment setting where new features may arise over time; e.g. a new attack-indicator starts being tracked at time t, where $\mathcal{F}^{(t)}$ depicts the feature space at time t. Specifically, at any time t, (1) new points $\mathbf{x}_{n+1}, \mathbf{x}_{n+2}, \ldots$ may arrive with dimensionality $|\mathcal{F}^{(t)}| \geq |\mathcal{F}^{(t-1)}|$, as well as (2) points seen thus far may receive value-updates to arbitrary (including new) features; where $< ID, F, \delta >$ denotes an update-triple for point with identifier ID to feature $F \in \mathcal{F}^{(t)}$ of value δ. For a real-valued feature F, $\delta \in \mathbb{R}$ is a value-update, whereas $\delta =$ `old_val:new_val` is a value-substitution for a categorical feature (`old_val` is `null` if F is a newly-arising feature). For example, `<id,URL,+3>` may indicate a social media user with identifier `id` sharing `3` more posts containing a link to a certain `URL`. Similarly, `<id,loc,NYC:Austin>` may depict a customer with `id` relocating (substituting `loc`) from `NYC` to `Austin`.

Vast majority of work for deployed OD systems assumes row-streams where newcoming points (i.e. rows) to be outlier-scored exhibit *fixed* dimensionality. To distinguish our deployed setting, where not only new rows but also new columns (i.e. features) may arrive, we refer to such incoming data as *evolving streams*.

2.1 Problem Statement

We aim to tackle the OD problem for very large datasets that are stored in a distributed fashion on commodity machines. This is typical of numerous settings where the data is prohibitively large to be stored on a single server or is already collected in a decentralized fashion (e.g. distinct customer bases around the globe).

We consider a *shared-nothing* parallel computing environment on which the data is to be processed, typical of many modern big data infrastructures including Apache Hadoop and Spark [6, 34]. Here each compute node (or worker machine) only has access to partial data, and processes it locally and independently of others(i.e. is idempotent), where intermediate results/data are then exchanged between the workers over the network. Computation typically alternates between several iterations of parallel local computation (e.g. a `map` phase) and communication (e.g. a `reduce` phase). Parallelism and local computation help with fast processing, whereas network speed is low, hence, network communication costs are often the bottleneck for distributed computing. Common strategies to reduce network costs include reducing the number of iterations (e.g. by computing locally more, and communicating less frequently) and/or reducing the size of the intermediate results to be communicated.

As we will show in Sec. 3, Sparx is only a *two-pass* algorithm, i.e. it requires only two iterations of `map` and `reduce` phases. Moreover, the intermediate data objects being passed between worker machines is of *constant size*, further reducing the burden (i.e. time-delay) of network communication.

We build Sparx for massive-scale cloud-resident data, although by design it can also handle distributed evolving streams. We present static and streaming problem definitions separately below. In Sec. 3 we describe the distributed algorithms underlying Sparx in detail, and discuss how to build on this solution to address streaming input when deployed in Sec. 3.5.

Problem 1 (Distributed OD for Static Data). ***Given*** *a static point-cloud database $\mathcal{P} = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ that is stored in a distributed file system (very large n);* ***Design*** *a distributed OD algorithm to compute outlier scores $\{s_1, \ldots, s_n\}$ on a shared-nothing computing infrastructure, with linear time and space complexity.*

Problem 2 (OD on Incoming Data Streams). ***Given*** *an incoming stream at $t = 1, 2, \ldots$, where (1) points with new ID may arrive with $|\mathcal{F}^{(t)}| \geq |\mathcal{F}^{(t-1)}|$ of mixed-type features, or (2) point-wise δ-updates $<ID, F, \delta>$ may arrive to existing points with ID and $F \in \mathcal{F}^{(t)}$;* ***Compute*** *(updated) outlier score for ID, in constant time.*

Before we delve into distributed algorithms, we provide a summary of the single-machine xStream OD algorithm in this section.

2.2 xStream for Single-Machine OD: Summary

xStream [20] is designed for outlier detection in high-dimensional data streams. In a nutshell, it consists of three main phases. First, to

[1] https://tinyurl.com/sparx2022

tackle high-dimensionality, it creates efficient data sketches, which can be computed on-the-fly even for newcoming features. Then, it builds efficient counting data structures for histogram-based density estimation in random subspaces of the feature space. Finally, it performs outlier scoring based on the approximated density estimates. We provide details on each of these steps as follows.

2.2.1 **Step 1. Data Projection.** Let us begin by assuming the feature space (i.e. dimensionality) is fixed, such that $\mathbf{x}_i \in \mathbb{R}^D$ where $D > d$ is the new dimensionality after one-hot-encoding (OHE) the categorical features. A low-dimensional sketch (or embedding) $\mathbf{s}_i$ can be created for each point (while accurately preserving pairwise distances between the points) by random projections [1, 15]:

$$\mathbf{s}_i = (\mathbf{x}_i^T \mathbf{r}_1, \ldots, \mathbf{x}_i^T \mathbf{r}_K) \tag{1}$$

where $\{\mathbf{r}_1, \ldots, \mathbf{r}_K\}$ depict K random Gaussian vectors [15] or *sparse* random vectors where with probability 1/3, $\mathbf{r}_k[F] \in \{\pm 1\}$ and zero otherwise [1]. The latter choice not only is "database-friendly", i.e. more efficient to store and compute, but can be also advantageous for outlier detection by effectively looking at data subspaces, reducing the masking effect of irrelevant features [35].

Notice that the same $\mathbf{r}_k \in \mathbb{R}^D$'s are used for all points over a stream, and hence need to be cashed. However, for evolving streams wherein new features may emerge, D is not fixed and in fact unknown apriori. Then, the idea is not to cash, but to hash. Specifically, entries of each $\mathbf{r}_k$ is to be computed on-the-fly via hashing, such that Eq. (1) is rewritten as follows.

$$\mathbf{s}_i[k] = \sum_{F \in \mathcal{F}_r} h_k(F) \cdot \mathbf{x}_i[F] + \sum_{F \in \mathcal{F}_c} h_k(F \oplus \mathbf{x}_i[F]) \cdot 1\,, \quad k = 1 \ldots K \tag{2}$$

where $h_k(\cdot)$ is a hash function, $\mathcal{F}_r$ and $\mathcal{F}_c$ denote the set of real-valued and categorical features, respectively, $\mathbf{x}_i[F]$ is point i's value of feature F, and $\oplus$ denotes the string-concatenation operator. Each hash function takes as input a string and returns $+1, -1$ or 0 with respective probabilities 1/6, 1/6 and 2/3. (See [20] for implementation details of such hash families.) For numerical features the input string is the feature name. For categorical features, it is the concatenation of the name and the corresponding feature value. Effectively, the sparse random vector entries are computed for any feature via hashing, i.e. $\mathbf{r}_k[F] = h_k[F]$, and multiplied with the corresponding feature value. For categorical features, the concatenated string corresponds to the OHE feature name with value 1.

When triplet updates $< ID, F, \delta >$ arrive over the stream, where δ = `old_val:new_val` for categorical features, the sketch can be updated by

$$\mathbf{s}_{ID}[k] = \begin{cases} \mathbf{s}_{ID}[k] + h_k(F) \cdot \delta & \text{if real-valued } F, \\ \mathbf{s}_{ID}[k] - h_k(F \oplus \texttt{old\_val}) + h_k(F \oplus \texttt{new\_val}) & \text{o.w.} \end{cases} \tag{3}$$

for $k = 1 \ldots K$ such that $h_k(F \oplus \texttt{old\_val})$ returns zero when `old_val` is `null`. It is important to notice that Eq. (3) can seamlessly handle a newly emerging feature F that has never been seen before.

2.2.2 **Step 2. Half-space Chains.** Anomaly detection relies on density estimation at multiple scales via a set of so-called Half-space Chains (HC), a data structure akin to multi-granular subspace histograms. Each HC has a length L (or L layers), along which the (projected) feature space $\mathcal{F}_p$ is recursively halved on a randomly sampled (with replacement) feature, where $f_l \in \{1, \ldots, K\}$ denotes the feature at level $l = 1, \ldots, L$. As such, a point can reside in one of 2 bins at level 1, one of 4 bins at level 2, and in general one of 2^l bins at level l. Given the sketch $\mathbf{s}$ of a point, the goal is to efficiently identify the bin it falls into at each level.

Let $\mathbf{\Delta} \in \mathbb{R}^K$ be the vector of initial bin widths, which is equal to half the range of the projected data along each dimension $f \in \mathcal{F}_p$. Let $\bar{\mathbf{z}}_l \in \mathbb{Z}^K$ denote the bin identifier of $\mathbf{s}$ at level l, initialized to all zeros. At level 1, the bin-id is updated as $\bar{\mathbf{z}}_1[f_1] = \lfloor \mathbf{s}[f_1]/\mathbf{\Delta}[f_1] \rfloor$. In general, the bin-id at consecutive levels can be computed *incrementally*, following

$$\bar{\mathbf{z}}_l[f_l] = \lfloor \mathbf{z}_l[f_l] \rfloor \;\text{ s.t. }\; \mathbf{z}_l[f_l] = \begin{cases} \mathbf{s}[f_l]/\mathbf{\Delta}[f_l] & \text{if } o(f_l, l) = 1, \text{ and} \\ 2\mathbf{z}_l[f_l] & \text{o.w.; if } o(f_l, l) > 1 \end{cases} \tag{4}$$

where $o(f_l, l)$ denotes the number of times feature $f_l = \{1, \ldots, K\}$ has been sampled in the chain until and including level l. We note that a small uniformly random value $\epsilon_l \in (0, \mathbf{\Delta}[f_l])$, called shift, is added to the sketches at each level to remedy issues for nearby points around fixed bin boundaries. We omit those for brevity and refer to [20] for details.

Notice that all points with the same unique $\bar{\mathbf{z}}_l$ reside in the same histogram bin at level l. As such, level-wise (multi-scale) densities are to be estimated by counting the number of points with the same bin-id at each level. This can be done by a dictionary (or perfect hash) data structure. The number of possible bins, however, grows exponentially with l. Even though data is not necessarily spread to all bins, number of non-empty bins (and hence the size of the dictionary) can grow very large for large L. Then, approximate counts can be obtained via a count-min-sketch [10], the size of which is user-specified, i.e. constant.

Overall, xSTREAM is an ensemble of M Half-Space Chains, $\mathcal{H} = \{HC^{(m)} := (\mathbf{\Delta}, \mathbf{f}^{(m)}, \boldsymbol{\epsilon}^{(m)}, \boldsymbol{C}^{(m)})\}_{m=1}^{M}$ where each HC is associated with the following list of meta-data; (i) the bin-width per feature $\mathbf{\Delta} \in \mathbb{R}^K$, (ii) the sampled feature per level $\mathbf{f}^{(m)} \in \mathbb{Z}^L$, (iii) the random shift value per level $\boldsymbol{\epsilon}^{(m)} \in \mathbb{R}^L$, and (iv) the counting data structure per level $C^{(m)} = \{C_l^{(m)}\}_{l=1}^{L}$.

2.2.3 **Step 3. Outlier Scoring.** To score a given (updated) sketch for outlierness, the count of points in the bin that it falls into is identified at each level l of a HC, denoted $C_l^{(HC)}[\bar{\mathbf{z}}_l]$. The count is extrapolated via multiplying by 2^l to estimate the total count if the data were distributed uniformly. Smallest estimate across levels[2] is taken as the outlier score, and then averaged across all HCs as

$$O(\mathbf{s}) = \frac{1}{M} \sum_{m=1}^{M} \min_l \; 2^l \cdot C_l^{(m)}[\bar{\mathbf{z}}_l]\,. \tag{5}$$

A lower value indicates a granularity at which the point resides in a relatively sparse region, and hence higher outlierness.

3 SPARX FOR DISTRIBUTED OD

In this section we introduce SPARX; distributed algorithms corresponding to each of the three main steps (as described in Sec.s §2.2.1–§2.2.3) of xSTREAM (Sec.s §3.1–§3.3), space and time complexity analysis (§3.4), and OD on incoming data streams (§3.5).

Our implementation uses Apache Spark's Python API. In Spark, the data points are stored in a distributed fashion across several compute nodes (or machines), residing in what-is-called a `DataFrame` (`DF`). The underlying distributed infrastructure is a shared-nothing

[2]Note that the counts can be compared across levels after extrapolation.

Algorithm 1 SPARX **Step 1.** Distributed Data Projection

Input: inputDF (input data), K (proj. dim.), featureNames
Output: projDF (K-dimensional transformed DF)

```
1: projector = HashProjn(K, seeds=arange(0, K, 1), density=1/3)
2: projDF = inputDF.map(lambda x: projector.fit_transform(x)
```

```
   procedure fit_transform(pt)
3:   for each F in featureNames:
4:     if F is categoric then F := F ⊕ pt[F]
5:   R = array([[hash_string(k, str) for k in seeds]
6:                                   for str in featureNames])
7:   return proj_pt = R · pt                    ▶ dot product
```

architecture, consisting of worker nodes and a driver node, which can be implemented with a MapReduce programming paradigm. The computation is broken down to pieces performed independently at worker nodes (typically through `map` operations), which may also exchange/communicate (over the network) intermediate data/results (typically through the `reduce` operation).

3.1 Step 1. Distributed Data Projection

The first step is to transform the input data to a K-dimensional representation through random projections. The projections are done based on Eq. (2) using random hash functions, $h_1(\cdot), \ldots, h_K(\cdot)$. As projection of each point i can be done independently using Eq. (2), the workers can perform this step *fully locally*, i.e. without any need for communication between workers. The projected data is stored in a new DF within the same worker nodes.

The pseudocode for Step 1 is given in Algorithm 1. We first define a `projector` with K different seeds between $(0, 1)$ and density 1/3 (Line 1). Note that the same seeds are used across all the worker nodes to create new points in the same embedding space. Operationally, projection step involves a *single* `map` *phase*. The `map` operator takes as input a function and passes each element of the DF through it. Namely the `fit_transform` function of the projector is fed to `map`, which transforms an input point by projecting it K times (Line 2). Steps of `fit_transform` is summarized in Lines 3–7, in which the `hash_string` function hashes the input string str (i.e. the feature name) based on the input seed, and with probability 1/3 returns $\{\pm 1\}$ and zero otherwise.

3.2 Step 2. Distributed Half-space Chains

After sparse projections we first obtain the feature ranges, specifically the gap between the minimum and maximum values in each of K features of the projected DF, and set the bin-widths $\boldsymbol{\Delta} \in \mathbb{R}^K$ to half of the ranges. The min and max values can be obtain easily for distributed data, by first finding those within each worker and then comparing the local min/max values across the workers. Next we start creating the half-space chains, as shown in Algorithm 2.

3.2.1 **Data-parallel training of a single chain.** Given $\boldsymbol{\Delta}$ and chain-length L (Line 1), each chain is instantiated at each level $l = 1 \ldots L$ with a randomly picked split-feature f_l from $\{1, \ldots, K\}$ as well as a random shift amount $\epsilon_l \in (0, \boldsymbol{\Delta}[f_l])$. These values are shared/common across the workers. Then the workers start binning the points, by computing the unique K-dimensional bin-id of each point at every level. Our implementation allows for constructing the histogram density estimation on a subsample of

Algorithm 2 SPARX **Step 2.** Distributed Half-space Chains

Input: projDF, L (chain-length), sampleRate, numRows, numCols, numChains, numThreads
Output: CMSketches (counts at all levels $l = 1 \ldots L$ per chain)

```
   procedure fit_chain(seed)
1:   C = Chain(Δ, L)
2:   binIDsDF = projDF.rdd.sample(sampleRate, seed)
3:                .map(lambda x: C.fit(x))
4:   cms = CMS(numRows, numCols)
5:   for l in range(L):
6:     C.CMSketches[l] = binIDsDF
7:       .flatMap(lambda x: cms.allCols(x[l]))
8:       .reduceByKey(lambda a,b : a+b) .collectAsMap()
```

```
9: tpool = ThreadPool(numThreads)
10: indxlist = list(range(numChains))
11: tpool.map(lambda cind: fit_chain(cind), indxlist)
```

the data (Line 2). As binning of points can be done independently, we implement this step through a `map` operation (Line 3), which passes each point through the `fit` function that returns the bin-id per level. The resulting bin-ids are stored in a new DF, called binIDsDF, in a distributed fashion.

Next we count the number of points that fall into each unique bin approximately, using a count-min-sketch (CMSketch) consisting of r (numRows) hash-tables, each with w (numCols) buckets (Line 4). Each level of a chain is associated with a separate CMSketch (Line 5–6). A CMSketch effectively hashes each input, i.e. a bin-id, into one of w buckets. Since collisions may occur, the hashing is repeated r times using different hash functions. The `allCols` function of the CMS class in our implementation takes a bin-id as input, and computes the bucket (i.e. column) index at each hash-table (i.e. row), returning an array of the form:

$$[((1, colindx\_1), 1), ((2, colindx\_2), 1), \ldots, ((r, colindx\_w), 1)] \quad (6)$$

As each bin-id can be hashed independently, we operationalize binning through a map, more specifically `flatMap`, which "flattens" the array of (`key`=(row,col), `value`=1) pairs returned by `allCols` into individual elements in a DF (Line 7). To sum up the count of points that hash into each bucket across workers, we perform a `reduceByKey`, which groups all pairs with the same `key` and sums their `values` (Line 8). Finally, `collectAsMap` gathers the total counts from across workers into the driver node (also Line 8).

3.2.2 **Model-parallel training of ensemble of chains.** As described, training of each chain leverages *data-parallelism* where we perform binning on partitions of data in parallel across worker nodes. Only the intermediate results, in this case partial sums/counts of points per bin are pooled over the network from all workers. SPARX is an *ensemble* of such half-space chains, each of which can be trained independently. In a single-machine implementation, these chains are built in sequence within a for-loop. To foster further speed up, we add to our implementation *model-parallelism* where the chains are trained by a pool of parallel threads (Lines 9–11).

3.3 Step 3. Distributed Outlier Scoring

Having collected all the partial counts across workers, the final CMSketches containing approximate bin counts reside in the driver node. For scoring, those are passed to individual workers which

Algorithm 3 SPARX **Step 3.** Distributed Outlier Scoring

Input: projDF, CMSketches (for all chains)
Output: outlier_scores

```
    procedure score_chain(cindex, CMS)
1:    C = Chains[cindex]
2:    scoreC=projDF.map(lambda x:C.score(x, CMS.value[cindex]))
3:  CMS = sc.broadcast(CMSketches)
4:  tpool = ThreadPool(numThreads)
5:  idx = list(range(numChains))
6:  outlier_scores = sum(tpool.map(lambda c: score_chain(c), idx))
```

can then *locally* compute the outlier score for each data point that they store. That is, outlier scoring involves a single `map` operation as outlined in Algorithm 3.

Being an ensemble, SPARX scores a point against each chain (Line 1). The main computation occurs during `map`, which passes each point through the `score` function of the chain to which we also feed as input argument the respective CMSketch containing the (approx.) bin counts at all levels $l = 1 \dots L$ (Line 2). Notably, we define CMSketches as a broadcast variable (Line 4), which tells Spark to pass it to workers *only once* with each `score` function call, since it is a fixed data structure that scoring does not alter.

For brevity, we do not include the steps for `score` in the pseudocode, which we briefly describe here in text. The steps are very similar to those for `fit` (i.e., identify the bin-id per level) and `allCols` (i.e., identify the bucket that bin-id hashes to per hash table, at each level). Each bucket is associated with a total count (held in CMSketch of the chain), which is an overestimate for the count of points with the same bin-id due to collisions. Therefore, the *minimum* count across the hash-tables is taken as the most accurate (i.e., least overestimate)—hence the name, count-*min*-sketch.

To obtain a point's outlier score per chain, the min-count at each level l is extrapolated by 2^l, and the smallest of the extrapolated counts is returned as the score from the current chain as in Eq. (5).

Similar to model-parallel fitting of the chains, we also score each point against the chains via a parallel thread pool (Lines 4–6). Differently, outlier score from each chain (i.e. thread) is summed across the pool (Line 6) and then averaged as in Eq. (5).

3.4 Space and Time Complexity

We analyze space and time complexity for each step of SPARX, and present storage and computation requirements both locally (per worker) and distributed (collectively across workers).

At the beginning, n d-dimensional data points are stored in a Spark DataFrame, taking $O(nd)$ distributed-storage. In **Step 1**, each feature is hashed K times using different seeds[3], where the resulting matrix R (Algo. 1, Line 5) takes $O(Kd)$ local-storage. Projecting one point is a dot product, with $O(Kd)$ complexity. Thus, Step 1 takes $O(Kdn)$ distributed-computation overall.

In **Step 2**, all M chains of the ensemble are trained in parallel by a thread pool, thus we multiply the all complexities below by M. The workers first compute the K-dim. bin-id of each point at each of L levels, hence the resulting binIDsDF takes $O(MKLn)$ distributed-storage. A bin-id per level is computed incrementally using Eq. (4), in $O(L)$ total time per point, and $O(MLn)$ distributed-computation. Next each point's bin-id is hashed r times at each level by the `allCols` function, for a total of $O(KrLMn)$ distributed-computation. The output is r pairs as in (6) per point per level, taking $O(MrLn)$ distributed-storage. The `reduceByKey` operation (Algo. 2, Line 8) groups all pairs with the same key=(row,col) in the same reducer node and sums the values, effectively finding the total count. This requires $O(MrLn)$ network communication, and $O(MrLn)$ distributed-computation.

There are at most r(numRows)×w(numCols) unique (row,col) keys which is equal to the user-specified CMSketch size. At the end, `collectAsMap` gathers these total counts across all layers and all chains at the driver node, requiring $O(rwLM)$ local-storage.

In **Step 3**, final CMSketches are passed from the driver to each worker for scoring, requiring $O(rwLM)$ local-storage. Scoring of a point has similar computational footprint to fitting where we, per level: create its bin-id, hash and read the counts from r buckets it hashes to, take the minimum and extrapolation. Minimum extrapolated count across L layers (i.e. outlier score) is averaged across M chains. Overall it takes $O(KrLMn)$ distributed-computation.

Remark: Note that SPARX is not only linear in data size (n and d) but also fully *data-parallel*—all bigO terms involving n are distributed. Moreover, all of K, L, r, w, M are user-specified, thus space and time associated with those can be adjusted on demand.

3.5 OD on Incoming Data Streams

Upon deployment, a single compute node can serve as the front-end to receive and score newcoming point updates over an evolving stream. It requires $O(rwLM)$ local-storage to keep all CMSketches in-memory. For each δ-update, the sketch- and then the score-update takes $O(K)$ and $O(KrLM)$, respectively. As existing points receive δ-update, a size-N LRU cache of IDs is maintained, along with their sketches for $O(NK)$ space. Note that both space and time complexity per update are constant, as all terms are user-specified.

4 EXPERIMENTS

4.1 Setup

4.1.1 **Datasets.** We experiment with three public-domain datasets, varying largely in number of points n and dimensions d, as well as in the fraction of outliers. Summary statistics are given in Table 1.

(1) *Gisette* is a handwritten digits dataset, originally from the UCI ML repository, which has also been used for outlier detection.[4] It has reasonably large number of features d, however its number of samples n is very small (in fact, smaller than d), at least for testing distributed algorithms. Following the outlier benchmark creation procedure by Steinbuss and Böhm [26], we fit a Gaussian Mixture Model (GMM) to the 3500 inliers it originally contains. Then we draw n=40,000 samples from the fitted GMM such that around 10% constitutes the outliers—inliers are drawn directly, while for generating outliers we increase the variance of 10% of the randomly chosen features by a factor of 5 as recommended in [26]. This ensures that 90% of the features do not convey any information on outlierness, making the detection task harder. We use the resulting dataset as a small-n/large-d testbed.

[3]Note that it is enough to hash the name of numerical features only once, whereas categorical feature names are concatenated by the value a point takes (Algo. 1, Line 4) and hence are (re)hashed per point.

[4]https://github.com/cmuxstream/cmuxstream-data

Table 1: Datasets used in experiments.

Name	n pts.	d dim.	size (GB)	type	outl.
Gisette	40,000	4,971	4.69	small-n/large-d	10%
OSM	2,772,233,904	2	51.50	large-n/small-d	0.036%
SpamURL	2,396,130	3,231,962		large-n/large-d	33%

(2) *OSM* [14] is a 2-d dataset depicting the GPS coordinates (latitude, longitude) collected from OpenStreetMap (OSM) contributors[5]. With nearly 3 billion points and 51.5GB total size, it is one of the largest real-world GPS datasets.[6] We use *OSM* as a very large-n/very small-d dataset. Besides its size, we use this low-dimensional dataset as it has been used for testing distributed OD algorithms previously [9, 32]. As we will show, these approaches are limited to such low-d settings and scale poorly with dimensionality.

The original *OSM* data does not contain any labeled outliers, and has been used in prior work [9, 32] as is, only for measuring runtime and scalability. We aim to study the complete landscape/trade-off between accuracy-versus-resources used. Therefore, to measure detection performance we inject into *OSM* simulated outliers, as described in detail in Appx. A.1.1. The visualization of the resulting dataset is shown in Fig. 1.

(3) *SpamURL* [19] is a large-n/very large-d dataset, which contains malicious and benign URLs along with numerous lexical and host-based characteristics of each URL as the features. This dataset makes the detection task challenging not only computationally, owing to its size, but also statistically as outliers are likely buried in small subspaces of the high dimensionality.

4.1.2 **Baselines.** Only a few OD algorithms are designed for distributed data and also have public-domain implementations. We compare Sparx to two such SOTA distributed OD methods.

(1) DBSCOUT [9]: This is the most recent distributed OD algorithm, implemented in Spark using the Java API[7]. It uses ideas that are largely inspired by the popular DBSCAN clustering algorithm [13], and constructs a cellular grid structure to parallelize and speed up the outlier identification process.

Another recent public-domain distributed OD approach, based on the popular LOF algorithm [8], is DDLOF [32]. We omit it from our baselines for two reasons; first, DBSCOUT has been shown to outperform DDLOF significantly on various datasets including our *OSM*, and second its implementation is in Hadoop[8] which is not on par with the efficiency of Apache Spark.

(2) SPIF [28]: Today, Spark is notably more popular than Hadoop providing orders of magnitude speed-up. Isolation Forest (IF) [16] is also one of the most popular OD algorithms owing to its competitive performance, as confirmed by evaluation studies [12, 29]. Therefore, we also compare to SPIF [28], a Spark-based design of IF, using a public-domain implementation.[9]

In a nutshell, IF subsamples data points to build a forest/ensemble of extremely randomized trees. In SPIF, each tree is trained in parallel at a single worker using its respective subsampled data. The subsample for each tree is gathered at a single worker via a map-reduce phase, where `<tree-ID, point>` pairs are generated during

[5]https://blog.openstreetmap.org/2012/04/01/bulk-gps-point-data/
[6]Data is downloaded from https://planet.osm.org/gps/simple-gps-points-120312.txt.xz
[7]https://github.com/mattecora/dbscout
[8]https://github.com/yizhouyan/DDLOFOptimized
[9]https://github.com/titicaca/spark-iforest

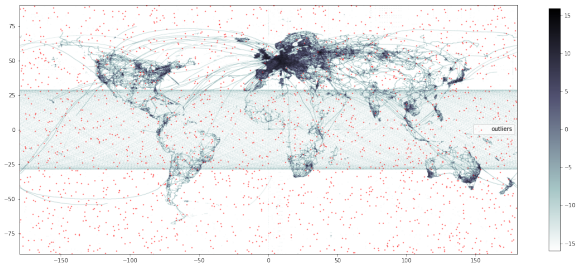

Figure 1: Visualization of the 2-d *OSM* dataset: (in black) inliers from real-world GPS traces, (in red) injected outliers.

`map`, and a `reduceByKey` is performed to shuffle all points needed to construct a tree to a single reducer/worker (!). As such, tree fitting is **not** data-parallel; rather, forest construction is designed to be model-parallel. For large subsample sizes, this implementation quickly becomes infeasible as it shuffles too much intermediate data over the network (which is very slow) prior to model fitting.

4.1.3 **Performance metrics.** To measure performance, we study the accuracy-vs-resource requirements landscape of the algorithms with varying HP configurations, since these typically have a trade-off relation. For accuracy, we report the *outlier ranking quality* using Area Under the ROC (AUROC), and Precision-Recall Curve (AUPRC), as well as F1 score. In terms of resources, we measure the *running time* and *peak memory usage*.

4.1.4 **System settings.** We conduct experiments on the U.S. National Science Foundation Pittsburgh Supercomputing Center (PSC), and set up both a 'moderate' (`config-mod`) as well as a 'generous' system configuration (`config-gen`) with relatively more resources, as specified in Table 5 in Appx.

Specifically, we vary the number of data (i.e. `DF`) partitions, available memory for the driver as well as the worker (executor) machines, the number of executors and number of cores per executor, as well as the number of threads available for multi-threading. `config-gen` has access to strictly more resources in all these aspects, typically doubling or more.

4.1.5 **Model settings.** We also run experiments with various hyperparameter (HP) configuration of the methods, as there is no clear means to setting those in unsupervised tasks. Our proposed Sparx is similar to SPIF in terms of being an ensemble of chains and trees, respectively, of certain depth, which can be trained on subsamples of data. Accordingly we vary the number of ensemble components M (tree or chain) ({50,100}), the depth or number of levels L ({10,20}), and subsampling rate ({0.01,0.1,1}) for Sparx and SPIF. We use a fixed CMS size of r=10×w=100 for Sparx on all datasets. The number of projections is set to K=50 for *Gisette*, and K=100 for *SpamURL*, while *OSM* is not transformed, since it is already very low dimensional.

On the other hand, DBSCOUT has two HPs; `eps` and `minPts`. We vary `minPts` and identify a corresponding `eps` via the process explained in [9], with quadratic (!), complexity: we plot the sorted distance to the `minPts`-th neighbor across all points. `eps` is then chosen in the uppermost part of the "elbow" zone of the plot.

4.2 Results

4.2.1 DBSCOUT **does not scale w.r.t. dimensionality** d**.** We start with analyzing results on our small-n/large-d *Gisette*. First,

Table 2: DBSCOUT **scales poorly with** d**. On** *Gisette*, **running time grows fast from** d**=1 to 10, and then times out.**

d dim.	Runtime (sec)	Peak memory (MB)
2	11.3	1,650
4	13.0	1,630
6	31.1	133,000
8	429.8	254,000
10	3,420.0	350,000
11	TIMEOUT	N/A

we show that DBSCOUT scales very poorly with increasing dimensionality using this moderately large dimensional dataset.

As shown in Table 2, when using `config-gen`, the running time grows dramatically fast as we run DBSCOUT on *Gisette* using an increasing number of randomly sampled 1–10 features. It reaches around 60-min mark at only d=10, and with 11 features, the process times-out (after 8 hrs). Peak memory usage responds similarly growing from 1.6GB to a total of 350GB across the executors.

Poor scalability makes DBSCOUT infeasible for datasets with more than a handful of features, therefore we only report results on *Gisette* for SPARX and SPIF.

4.2.2 **Accuracy vs. Resources Landscape.** Next we analyze the trade-off between detection quality and resources required. Fig. 2 shows AUROC (y-axis) versus total running time (x-axis, left) and peak driver memory (x-axis, right) under `config-gen`. We find that SPIF performance varies between 0.72-0.80 across HP configurations whereas SPARX reaches 0.80-0.87. On the other hand, SPARX uses more resources; as compared to SPIF's typical runtime 1-2 minutes, SPARX can achieve its peak performance in around 14 minutes, using 2-3× more memory. Similar conclusions are drawn under `config-mod`, which is in Appx. A.2.1, Fig. 7.

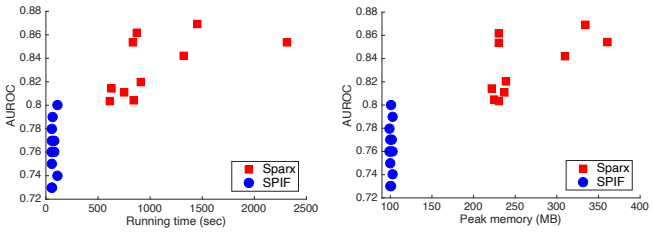

Figure 2: Comparing SPARX (red) and SPIF (blue) on *Gisette*: **(left) Running time (sec) vs. accuracy in AUROC, and (right) Peak driver memory (MB) vs. AUROC. Symbols depict different hyperparameter configurations.** DBSCOUT **does not run on** *Gisette* **due to dimensionality.**

Table 3 presents a more "head-to-head" comparison under comparable settings of the HPs for both methods. Increasing (doubling) the number of ensemble components improves SPARX's accuracy, with no effect on SPIF; whereas it is vice versa when increasing the sampling rate. However, even with all data (sampling rate = 1), SPIF does not exceed an AUROC of 0.8.

Importantly, notice that when the number of samples per tree goes up from 4,000 (rate=0.1) to 40,000 (rate=1), SPIF's running time notably increases. This is due to its model-parallel yet **not** data-parallel nature; where all samples per tree are shuffled (i.e. copied) over the network to a worker that is designated to construct the tree. With 100 trees, the total network communication becomes

Table 3: "Head to head " comparison of SPARX and SPIF on *Gisette* **under equivalent hyperparameter configurations.**

				AUROC		Time(s)		Mem (MB)	
conf.	#comp.	sampl.	depth	Sx	DIF	Sx	DIF	Sx	DIF
1	50	0.01	10	0.80	0.77	610.5	58.0	230.2	99.6
2	100	0.01	10	0.85	0.76	836.0	58.0	230.7	99.0
3	100	0.1	10	0.86	0.79	874.2	61.1	229.9	102.8
4	100	0.1	20	0.87	0.78	1455.6	60.1	334.5	99.0
5	100	1	20	0.85	0.80	2312.8	111.0	360.8	101.1

notable. In fact, SPIF quickly becomes infeasible to run on massive datasets even with a tiny subsampling rate, exactly due to this data shuffling problem, as we show in the next section.

4.2.3 SPIF **does not scale w.r.t. input size** n**.** As a reminder, a key design principle in distributed computing on big data is "*code goes to data*" and **not** vice versa. In other words, the goal is to compute as much as possible *locally* and not to move around data between compute nodes (other than data containing intermediate results). The way SPIF is implemented violates this principle and suffers on large-scale data. We demonsrate the problem using *OSM*.

Using `config-gen`, we input to SPIF[10] a gradually increasing fraction of *OSM* (for *fitting*, while all 2.7+ billion points are scored), starting only with around 1/1000'th of the data, as shown in Table 4. As we double the input size at every round, total running time and memory usage increase accordingly. However, when the number of points per tree reaches around half a million, we get a system memory error and the program crashes. As we continue to increase data size, data processing cannot reach the memory error before the 8-hour SC-budget is exhausted and we get a system time-out.

Table 4: SPIF **does not scale up w.r.t. input size** n**.**

Frac.	#pts/tree	Time (s)	Mem (GB)	AUPRC	AUROC
0.00128	35,471	1396	454	0.19	0.987
0.00256	70,943	1402	455	0.27	0.989
0.00512	141,887	1531	461	0.38	0.991
0.01024	283,774	1834	463	0.42	0.993
0.02048	567,548	MEM ERR	-	-	-
0.04096	1,135,097	MEM ERR	-	-	-
0.08192	2,270,194	TIMEOUT	-	-	-
0.16384	4,540,389	TIMEOUT	-	-	-

As we have done, it is possible to fit SPIF on a small subsample of a massive dataset – to avoid this error during fitting – and still be able to *score all* data points. However, as Table 4 shows this incurs a sacrifice in detection performance (note esp. the AUPRC).

4.2.4 **Large-**n**/Small-**d**.** Next we analyze the results on *OSM*, containing billions of points but only 2 dimensions. Fig. 3 shows the landscape of detection performance (F1) vs. resources used for all three methods, under varying HP configurations. (See Appx. A.2.2, Tables 7–10 for detailed numbers.) Note that DBSCOUT outputs a binary label, thus we can only report F1 for comparison.[11]

SPIF can be fit using at most 10^{-4}-th of the data, as discussed earlier, which results in very poor performance (F1<0.2). DBSCOUT is the fastest on this low-d setting and achieves the most competitive performance, however, it is quite sensitive to the HP choices and its performance oscillates widely. SPARX performance is more stable, with a longer processing time, while using less memory.

[10] We set model HPs as num_trees=50, max_depth=25, and sample_rate=0.01.
[11] Detailed numbers in Appx. include AUROC and AUPRC for SPIF and SPARX.

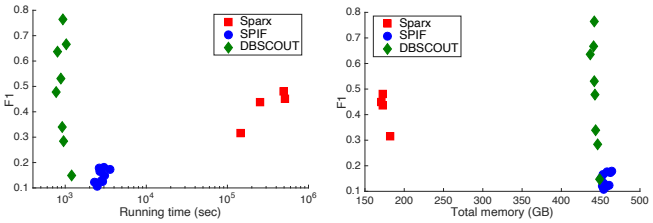

Figure 3: Comparing all methods on *OSM* under `config-gen`: (left) Running time (sec) vs. accuracy in F1, and (right) Total memory (GB) vs. F1. Symbols depict different HP config.s.

4.2.5 **Large-*n*/Large-*d*.** Finally, we present a similar analysis on our very large-*d* (yet sparse) *SpamURL*. Problematically, the SPIF implementation cannot handle sparse RDD input (and it is infeasible to store *SpamURL* as a dense RDD). Therefore, we transform it using our random projections to d=100. (Our Sparx also uses K=100 projections.) DBSCOUT scales very poorly with dimensionality, for which we also transform *SpamURL* to d=7 (largest d that DBSCOUT could handle), as well as d=2 (for comparison).

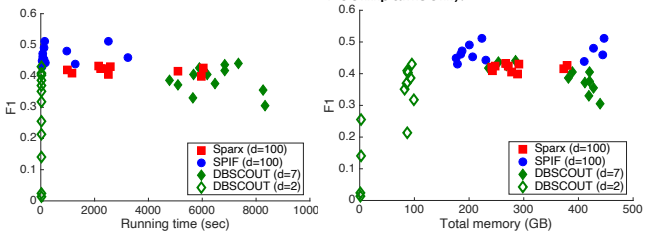

Figure 4: Comparing methods on *SpamURL* under `config-gen`: (left) Running time (sec) vs. accuracy in F1, and (right) Total memory (GB) vs. F1. Symbols depict different HP config.s.

As shown in Fig. 4, DBSCOUT with d=2 is quite resource-frugal, however it has a widely varying performance depending on the choice of hyperparameters. It achieves more stable performance with d=7, yet is inferior to SPIF in both time and accuracy. On the other hand, Sparx performance is robust to different hyperparameter settings, and is on par with the competing baselines.

4.2.6 **Speed-up by increasing parallelism.** Next we show how Sparx leverages data-parallelism, Using *Gisette*, we increase the number of DataFrame partitions on Spark. Fig. 5 shows that the running time decreases as partitions increase from 8 to 128, and then slightly increases for 256. This is expected behavior of distributed platforms—that speed-up is not monotonic: when the data is partitioned too much to the extent that each worker is under-utilized, the cost of network communication between workers overtakes and reduces the gains from parallelism. As compared to the running time of single-machine xStream, Sparx provides 4-20× speed-up.

4.2.7 **Sparx scalability with input size *n*.** Finally, we study the scalability of our distributed Sparx w.r.t. input size. Recall that dimensionality d is associated with Step 1. (projection) of the algorithm, where in Sec. 3.4 we have shown that Sparx is linear in d. We have also shown that across all 3 steps, it is linear in the number of data points n. Since Spark-like platforms are distributed/data-parallel in n, we study the running time of Sparx for increasing sizes of n using *OSM*.[12] As shown in Fig. 6, Sparx scales linearly w.r.t. n, emprically confirming our complexity analysis in Sec. 3.4.

[12]We set model HPs as num_chains=10, depth=5, and sample_rate=1.

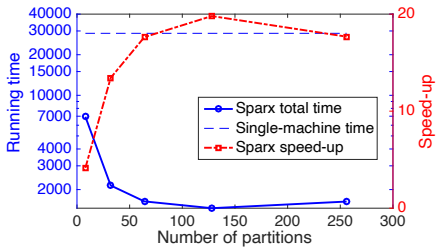

Figure 5: (in blue) Running time of Sparx on *Gisette* as number of data partitions increases. (in red) Speed-up w.r.t. single-machine implementation.

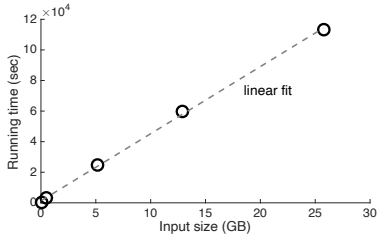

Figure 6: Sparx scales linearly in number of points *n*.

Summary and remarks. To sum up, through extensive experiments we showed that Sparx applies to all large-n and/or large-d datasets, provides competitive detection accuracy-running time trade-off, takes advantage of data parallelism effectively, and scales up with increasing input size. In contrast, among the handful of public-domain distributed OD algorithms, (1) DBSCOUT **has poor scalability w.r.t. dimensionality** d, being applicable to only small-d (d<10) datasets in practice. On the other hand, (2) SPIF **cannot handle large-n datasets** due to its non-data-parallel implementation, also rendering it a non-practical choice.

5 RELATED WORK

Outlier mining has a large literature owing to its many high-stakes applications in finance, environmental monitoring, surveillance and security, to name a few. However, most existing work on point outlier detection (OD) [2], including those for data streams [20, 21], are designed for a single machine. Distinctly, we focus our survey on distributed detection techniques.

Although OD for large-scale data is extremely important in the big data context, and likely to become more relevant over time, there are relatively much fewer *parallel* OD algorithms for truly *distributed* environments with thousands of compute nodes, such as cloud services. A group of parallel algorithms are designed for *shared-memory* multi-core computer systems [17, 23] and not for distributed settings. Other parallel algorithms for distributed architectures are centralized; requiring a central "communication/sync" unit. For example, the top-n OD algorithms by Angiulli *et al.* [4, 5] assume a "supervisor" node for synchronization. Similarly, Bhaduri *et al.* [7] also require a "central" node that maintains and updates the top-n points. Those are not applicable to modern scale-out (i.e. distributed) *shared-nothing* architectures that do not employ such centralization. Moreover, those work focus on conceptual algorithm design and do not present practical implementations.

We remark that a related category of work on distributed OD for wireless sensors [18, 24, 30] is notably different from our work, in that those often require communication between (nearby) sensors

and operate under battery/power constraints which do not apply to shared-nothing settings.

Among distributed OD algorithms for shared-nothing architectures, Tao *et al.* proposed SPIF [28], a Spark-based design of the popular Isolation Forest algorithm [16]. However, SPIF emloys *model-parallelism* (as opposed to data-parallelism); specifically it trains each individual component (i.e. iTree) of the IF ensemble on a separate compute node. Alarmingly, the data for each iTree is shuffled to the corresponding node over the network, adding to the communication cost. Despite radically growing real-world datasets with billions of points, to the best of our knowledge, there are only two *data-parallel* (i.e., horizontally scalable [9]) distributed OD solutions in the literate thus far.

Yan *et al.* proposed DDLOF [32], the first distributed LOF algorithm for Hadoop MapReduce [11], and later its extension to top-n outliers [33]. Besides Hadoop being orders of magnitude slower than Spark [34], as has been shown recently [9], DDLOF fails to scale to very large datasets. Their grid-based data partitioning strategy makes it unsuitable for high-dimensional data as the number of partitions grows exponentially with increasing dimensionality.

Most recently, Corain *et al.* introduced DBSCOUT [9] with a public-domain Spark implementation. By design, it does not provide a ranking of the outliers (i.e. output is binary) and has two critical hyperparameters (`eps` and `minPts`) to set. From a scalability perspective, even though it is linear in the number of input points, it scales extremely poorly with dimensionality d. All of their experiments are limited to 2- or 3-d datasets.

6 CONCLUSION

We presented SPARX, a new scalable open-source tool for distributed outlier detection (OD). We described its design principles and the underlying distributed/data-parallel algorithms for shared-nothing cloud-computing platforms, and open-sourced its Apache PySpark implementation at https://tinyurl.com/sparx2022.

OD finds numerous applications, yet there are limited public-domain resources for *distributed* OD as the vast majority of the literature focuses on single-machine algorithmic problems. Through extensive experiments, we showed that the few existing open-source tools do not match up with SPARX; they either do not scale well with dataset size or increasing dimensionality. Distinctly, SPARX is fully data-parallel, and scales linearly. We believe SPARX sets the state-of-the-art in terms of detection performance and scalability for distributed OD tasks. We expect it to increase the usability of OD on large-scale modern-day datasets that are already cloud-resident, and to offer significant impact in the applied domain for various business, engineering and scientific use cases.

ACKNOWLEDGMENTS

This work is sponsored by the U.S. Army Network Enterprise Technology Command (NETCOM) and NSF CAREER 1452425. Any conclusions expressed in this material are those of the author and do not necessarily reflect the views of the funding parties.

REFERENCES

[1] Dimitris Achlioptas. 2003. Database-friendly random projections: Johnson-Lindenstrauss with binary coins. *J. of Comp. and Sys. Sci.* 66, 4 (2003), 671–687.
[2] Charu C Aggarwal. 2017. *Outlier analysis.* Springer.
[3] Archana Anandakrishnan, Senthil Kumar, Alexander Statnikov, Tanveer Faruquie, and Di Xu. 2018. Anomaly detection in finance. In *KDD Workshop on ADF*. 1–7.
[4] Fabrizio Angiulli, Stefano Basta, Stefano Lodi, and Claudio Sartori. 2010. A distributed approach to detect outliers in very large data sets. In *European Conference on Parallel Processing*. Springer, 329–340.
[5] Fabrizio Angiulli, Stefano Basta, Stefano Lodi, and Claudio Sartori. 2012. Distributed strategies for mining outliers in large data sets. *TKDE* 25, 7 (2012).
[6] Apache Software Foundation. [n.d.]. *Hadoop.* https://hadoop.apache.org
[7] Kanishka Bhaduri, Bryan L Matthews, and Chris R Giannella. 2011. Algorithms for speeding up distance-based outlier detection. In *KDD*. 859–867.
[8] Markus M Breunig, Hans-Peter Kriegel, Raymond T Ng, and Jörg Sander. 2000. LOF: identifying density-based local outliers. In *SIGMOD*. 93–104.
[9] Matteo Corain, Paolo Garza, and Abolfazl Asudeh. 2021. DBSCOUT: A Density-based Method for Scalable Outlier Detection in Very Large Datasets. In *ICDE*. 37–48.
[10] Graham Cormode and Shan Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. *J. of Alg.* 55, 1 (2005).
[11] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.
[12] Andrew Emmott, Shubhomoy Das, Thomas Dietterich, Alan Fern, and Weng-Keen Wong. 2015. A meta-analysis of the anomaly detection problem. *arXiv preprint arXiv:1503.01158* (2015).
[13] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise.. In *KDD*, Vol. 96. 226–231.
[14] Mordechai Haklay and Patrick Weber. 2008. Openstreetmap: User-generated street maps. *IEEE Pervasive Comp.* 7, 4 (2008), 12–18.
[15] Piotr Indyk and Rajeev Motwani. 1998. Approximate nearest neighbors: towards removing the curse of dimensionality. In *STOC*. 604–613.
[16] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. 2008. Isolation forest. In *ICDM*. IEEE, 413–422.
[17] Elio Lozano and Edgar Acufia. 2005. Parallel algorithms for distance-based and density-based outliers. In *ICDM*. IEEE, 4–pp.
[18] Tie Luo and Sai G Nagarajan. 2018. Distributed anomaly detection using autoencoder neural networks in WSN for IoT. In *ICC*. IEEE, 1–6.
[19] Justin Ma, Lawrence K. Saul, Stefan Savage, and Geoffrey M. Voelker. 2009. Identifying suspicious URLs: an application of large-scale online learning.. In *ICML*. ACM, 681–688.
[20] Emaad Manzoor, Hemank Lamba, and Leman Akoglu. 2018. xStream: Outlier detection in feature-evolving data streams. In *KDD*. 1963–1972.
[21] Gyoung S Na, Donghyun Kim, and Hwanjo Yu. 2018. Dilof: Effective and memory efficient local outlier detection in data streams. In *KDD*. 1993–2002.
[22] Jose M Navarro, Alexis Huet, and Dario Rossi. 2021. Human readable network troubleshooting based on anomaly detection and feature scoring. *arXiv preprint arXiv:2108.11807* (2021).
[23] Junki Oku, Keiichi Tamura, and Hajime Kitakami. 2014. Parallel processing for distance-based outlier detection on a multi-core CPU. In *IWCIA*. IEEE, 65–70.
[24] Themistoklis Palpanas, Dimitris Papadopoulos, Vana Kalogeraki, and Dimitrios Gunopulos. 2003. Distributed deviation detection in sensor networks. *SIGMOD* 32, 4 (2003), 77–82.
[25] Angela A Sodemann, Matthew P Ross, and Brett J Borghetti. 2012. A review of anomaly detection in automated surveillance. *Trans. on Sys., Man, and Cybernetics* 42, 6 (2012), 1257–1272.
[26] Georg Steinbuss and Klemens Böhm. 2021. Benchmarking Unsupervised Outlier Detection with Realistic Synthetic Data. *ACM TKDD* 15, 4 (2021), 1–20.
[27] Gian Antonio Susto, Matteo Terzi, and Alessandro Beghi. 2017. Anomaly detection approaches for semiconductor manufacturing. *Procedia Manufacturing* 11 (2017), 2018–2024.
[28] Xiaoling Tao, Yang Peng, Feng Zhao, Peichao Zhao, and Yong Wang. 2018. A parallel algorithm for network traffic anomaly detection based on Isolation Forest. *International Journal of Distributed Sensor Networks* 14, 11 (2018).
[29] Kai Ming Ting, Takashi Washio, Jonathan R Wells, and Sunil Aryal. 2017. Defying the gravity of learning curve: a characteristic of nearest neighbour anomaly detectors. *Machine Learning* 106, 1 (2017), 55–91.
[30] Yu-Lin Tsou, Hong-Min Chu, Cong Li, and Shao-Wen Yang. 2018. Robust distributed anomaly detection using optimal weighted one-class random forests. In *ICDM*. IEEE, 1272–1277.
[31] Xiang Xu, Yuan Ren, Qiao Huang, Zi-Y Fan, Zhao-J Tong, Wei-J Chang, and Bin Liu. 2020. Anomaly detection for large span bridges during operational phase using structural health monitoring data. *Smart Mater. and Struc.* 29, 4 (2020).
[32] Yizhou Yan, Lei Cao, Caitlin Kulhman, and Elke Rundensteiner. 2017. Distributed local outlier detection in big data. In *KDD*. 1225–1234.
[33] Yizhou Yan, Lei Cao, and Elke A Rundensteiner. 2017. Distributed Top-N local outlier detection in big data. In *BigData*. IEEE, 827–836.
[34] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*.
[35] Arthur Zimek, Erich Schubert, and Hans-Peter Kriegel. 2012. A survey on unsupervised outlier detection in high-dimensional numerical data. *Stat. Analysis and Data Mining* 5, 5 (2012), 363–387.

A APPENDIX

A.1 Details on Experiment Setup

A.1.1 Ground-truth outliers in OSM. In previous work [9, 32], the OpenStreetMap (*OSM*) dataset was used only for scalability and running time experiments but not for evaluation detection performance, since it does not contained labeled outliers. In this work, we inject simulated outliers in order to evaluate and compare algorithms in all respects; detection quality, as well as resources (time and memory) used.

How we define and simulate these ground-truth outliers are as follows. Originally, the *OSM* dataset contains 2,771,233,904 points of GPS coordinates, i.e. tuples of latitude and longitude values, from real-world users' travel trajectories. To inject outliers, we first generate a 2-d histogram of the full dataset. This is done by first creating a grid with cell size (0.01×0.01) that covered the full space (-180,180) × (-90,90). Then, we count the number of points that fall into each cell and mark all empty grid cells whose immediate 8 neighbours are also empty. Each outlier was then generated by randomly picking one of these marked cells and then uniformly selecting coordinates within the cell. Our final dataset consists of 2,772,433,904 billion points, of which 1,000,000 (0.036%) are outliers.

The visualization of the final dataset is shown in 1, where black dots depict GPS coordinates visited by real-world users, and red dots illustrate the injected outliers.

A.1.2 System configuration details. Specified in Table 5.

Table 5: Two different system config.s used in experiments; 'moderate' `config-mod` and more 'generous' `config-gen`.

	#partitions	driver memory	exec memory	#execs	#exec cores	#threads
`config-mod`	64	25GB	4GB	4	4	4
`config-gen`	128	45GB	8GB	64	8	128

A.2 Additional/Detailed Experiment Results

A.2.1 **Results on *Gisette* under `config-mod`.** In Fig. 7 We showcase the AUROC performance versus running time (left) and peak memory (right) on *Gisette* under the moderate system configuration `config-mod`. Results are for Sparx and SPIF only, since DBSCOUT cannot scale beyond about 10 dimensions.

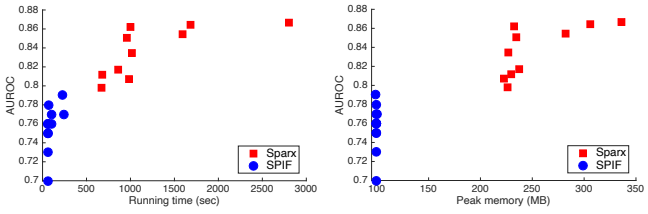

Figure 7: Using `config-mod`: (left) Running time (sec) vs. accuracy in AUROC, and (right) Peak driver memory (MB) vs. AUROC on *Gisette* comparing Sparx (red) and SPIF (blue). DBSCOUT does not run on *Gisette* due to dimensionality.

A.2.2 **Detailed results on *OSM*.** Tables 6 and 7 (under `config-mod` and `config-gen`, respectively) provide performance and resource usage details of SPIF on *OSM* with varying hyperparameter settings. Notice that we subsample *OSM* substantially for being able to fit SPIF, as not being data-parallel, it does not scale well with n.

Table 6: SPIF performance and resources used on *OSM* under `config-mod` and varying HP configurations. Note that fewer config.s can be handled under moderate resources as compared to `config-gen`. Sampling rate is from 1% of the original data, and thus should be multiplied further by 0.01×.

#comp.	depth	sampl.	Time(s)	Mem(GB)	AUROC	AUPRC	F1
50	10	0.001	2284	478	0.978	0.161	0.115
50	10	0.005	2562	481	0.980	0.327	0.164
50	20	0.001	2530	481	0.986	0.149	0.118
50	20	0.005	2908	486	0.991	0.389	0.162
100	10	0.001	2920	480	0.979	0.144	0.118
100	20	0.001	3475	483	0.986	0.155	0.123

Table 7: SPIF performance and resources used on *OSM* under `config-gen` and varying hyperparameter (HP) configurations. Sampling rate is from 1% of the original data, and thus should be multiplied further by 0.01×.

#comp.	depth	sampl.	Time(s)	Mem(GB)	AUROC	AUPRC	F1
50	10	0.001	2292	452	0.978	0.139	0.122
50	10	0.005	2662	453	0.980	0.311	0.165
50	10	0.010	2474	457	0.978	0.346	0.119
50	20	0.001	2463	454	0.986	0.150	0.107
50	20	0.005	2627	458	0.991	0.367	0.177
50	20	0.010	2975	464	0.992	0.439	0.180
100	10	0.001	2844	454	0.978	0.166	0.131
100	10	0.005	3076	450	0.979	0.324	0.150
100	20	0.001	2913	461	0.986	0.159	0.124
100	20	0.005	3555	462	0.991	0.387	0.174

Tables 8 and 9 (under `config-gen` and `config-mod`, respectively) provide performance and resource usage details of DBSCOUT on *OSM* with varying hyperparameter settings. DBSCOUT excels on this very low dimensional dataset, as it is designed accordingly.

Table 8: DBSCOUT performance and resources used on *OSM* under `config-gen` and varying HP configurations. Note that DBSCOUT output is binary and thus only F1 is reported.

minPts	eps.	Time(s)	Mem(GB)	F1
100	250000	957	446	0.283
100	500000	775	443	0.478
100	1000000	799	437	0.637
100	2000000	938	442	0.765
200	250000	1209	449	0.148
200	500000	918	444	0.339
200	1000000	884	442	0.531
200	2000000	1030	441	0.667

Table 9: DBSCOUT performance and resources used on *OSM* under `config-mod` and varying hyperparameter configurations. Note that DBSCOUT output is binary and thus only F1 is reported. DBSCOUT scales well to this large-n/small-d (2-d) dataset, w/ comparable results to those under `config-gen`.

minPts	eps.	Time(s)	Mem(GB)	F1
100	250000	1279	474	0.283
100	500000	911	469	0.478
100	1000000	855	468	0.637
100	2000000	1167	466	0.764
200	250000	1615	478	0.148
200	500000	1031	468	0.339
200	1000000	930	466	0.531
200	2000000	1257	468	0.667

Table 10: SPARX performance and resources used on *OSM* under `config-gen` and varying hyperparameter configurations (sampling rate is set to 0.01).

#comp.	depth	Time(s)	Mem(GB)	AUROC	AUPRC	F1
10	5	144041	182.36	0.959	0.271	0.316
10	10	254243	172.89	0.973	0.400	0.437
20	10	509014	170.72	0.974	0.443	0.451
10	20	492506	172.85	0.975	0.446	0.480

A.2.3 **Detailed results on *SpamURL* .** Table 11 provides performance and resource usage details of SPIF on *SpamURL* with varying hyperparameter settings under `config-mod`.

Note that *SpamURL* is very high dimensional yet sparse, however the SPIF implementation cannot handle sparse RDD input (and it is infeasible to store *SpamURL* as a dense RDD). Therefore, we transform it using our random projections to d=100. (Our SPARX also uses K=100 projections.)

Table 11: SPIF performance and resources used on *SpamURL* (d=100) under `config-mod` and varying hyperparameter configurations. The best (in bold) and the worst F1 performance highlighted (only measure DBSCOUT can be compared to).

#comp.	depth	sampl.	Time(s)	Mem(GB)	AUROC	AUPRC	F1
50	10	0.01	61.8	206	0.656	0.479	0.463
50	10	0.1	136.0	279	0.703	0.524	**0.526**
50	20	0.01	63.0	201	0.684	0.502	0.488
50	20	0.1	128.8	285	0.677	0.491	0.468
100	10	0.01	80.6	202	0.689	0.503	0.498
100	10	0.1	150.8	318	0.676	0.484	0.475
100	20	0.01	83.0	204	0.639	0.457	0.434
100	20	0.1	159.5	328	0.659	0.475	0.451
50	10	1	938.8	451	0.675	0.492	0.481
50	20	1	1192.5	440	0.637	0.461	0.439
100	10	1	2520.1	440	0.637	0.461	0.439

Tables 12 and 13 (under d=7 and d=2, respectively) provide performance and resource usage details of DBSCOUT on *SpamURL* with varying hyperparameter settings under `config-mod`. (As a simple heuristic, we set `minPts` to $2\times d$ and then carefully choose `eps` via the elbow-method as explained in [9].)

As a reminder, DBSCOUT scales very poorly with dimensionality and cannot handle the original *SpamURL* dataset. Therefore, as with SPIF, we reduce dimensionality via random projections. The largest d that DBSCOUT was able to handle is d=7, and we also report results with d=2 for comparison. Resources (time and memory) reduce for the latter, however at the expense of detection performance.

Table 14 provides performance and resource usage details of SPARX on *SpamURL* with varying hyperparameter settings under `config-mod`. Note that its performance is quite stable/robust to varying hyperparameter choices.

Table 12: DBSCOUT performance and resources used on *SpamURL* (d=7) under `config-mod` and varying hyperparameter configurations. The best (in bold) and the worst performance highlighted.

minPts	eps.	Time(s)	Mem(GB)	F1
14	0.6	9589	233	0.418
14	0.7	8860	276	0.426
14	0.8	7739	385	0.386
14	0.9	12424	419	0.371
14	0.95	5697	430	0.330
14	1	15616	438	0.306
28	0.6	10272	235	0.437
28	0.7	8591	284	**0.441**
28	0.8	9269	382	0.406
28	0.9	12000	421	0.405
28	1	8221	432	0.356

Table 13: DBSCOUT performance and resources used on *SpamURL* (d=2) under `config-mod` and varying hyperparameter configurations. The best (in bold) and the worst performance highlighted. Notice that with only d=2, DBSCOUT performs notably worse than that for d=7, while in turn, correspondingly lower resources are required.

minPts	eps.	Time(s)	Mem(GB)	F1
4	0.0001	476	116	0.410
4	0.0005	429	130	0.370
4	0.001	559	2.2	0.352
4	0.005	1139	1.83	0.256
4	0.01	1158	1.97	0.141
4	0.05	538	1.37	0.013
8	0.0001	854	126	**0.431**
8	0.0005	544	118	0.403
8	0.001	1139	129	0.386
8	0.005	1129	129	0.318
8	0.01	817	1.27	0.213
8	0.05	201	1.7	0.023

Table 14: SPARX performance and resources used on *SpamURL* (K=100) under `config-mod` and varying hyperparameter configurations. The best (in bold) and the worst F1 performance highlighted (only measure DBSCOUT can be compared to).

#comp.	depth	sampl.	Time(s)	Mem(GB)	AUROC	AUPRC	F1
50	10	0.01	980.5	241	0.602	0.419	0.420
50	10	0.1	1150.1	243	0.590	0.407	0.410
50	20	0.01	2160.2	267	0.620	0.43	**0.433**
50	20	0.1	2523.0	277	0.595	0.409	0.406
100	10	0.01	2324.1	271	0.613	0.424	0.423
100	10	0.1	2584.8	290	0.617	0.429	0.430
100	20	0.01	5089.6	372	0.600	0.419	0.416
100	20	0.1	6036.6	379	0.614	0.428	0.426
50	10	1	2223.5	247	0.609	0.421	0.424
50	20	1	5965.4	288	0.577	0.403	0.399