Streaming and Distributed Anomaly Detection and its Applications

Heek, H.B. (Ruben)

Award date:
2022

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Department of Mathematics and Computer Science
Algorithms, Geometry, and Applications Research Group

Streaming and Distributed Anomaly Detection and its Applications

Master's Thesis

Ruben Heek

Supervisors:
Dr. Marcel J.M. Roeloffzen
Dr. rer. nat. Morteza Monemizadeh
Dr. Yulong Pei

Eindhoven, September 2022

Abstract

Anomaly detection is the problem of finding unexpected patterns in data. With the growing size of data being generated, it becomes increasingly difficult to detect anomalies, especially when the data: (i) is given in a streaming fashion and needs to be processed in real-time or using small space; (ii) is distributed among many many machines for which the communication bottleneck ought to be reduced. This thesis proposes the use of Random Shift Forest (RSF), a simple yet powerful isolation-based anomaly detector, and discusses how it can be extended to a wide variety of models and data. Various versions of streaming and distributed RSF are formalised, and their performance is analysed and empirically verified. In doing so, this thesis develops a broadly applicable anomaly detection framework to use and build upon.

Preface

In the first place, I would like to thank my supervisor for this Master's thesis, Morteza Monemizadeh, for starting this project with me and supervising me throughout its whole duration. His ideas and suggestions, and all other friendly conversations, provided the necessary help to bring it to the complete work presented here. Of course, I also want to thank the other members of the defence committee, Marcel Roeloffzen and Yulong Pei, for their time and energy in being part of the evaluation of this thesis and the defence. Last but not least, I want to thank my family and friends for all the support, food, coffee, and distraction.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Random Shift Forest (RSF) was first introduced by Thijs Visser in his master's thesis [65]. RSF was designed to be implemented in the streaming and distributed settings. Visser was only able to implement it in the offline setting. He benchmarked RSF on several toy and real datasets and showed it to outperform Isolation Forest [45]; a state-of-the-art technique for unsupervised anomaly detection in high-dimensional Euclidean spaces. He also presented several arguments for why RSF could be a good candidate for detecting anomalies in the streaming and distributed settings. No actual implementations were provided for these, however.

This thesis builds upon Visser's work by exploring a range of extensions and applications for the RSF algorithm. In particular, an extension of RSF is proposed that can be implemented in the streaming and distributed settings. This extended version is then applied to detect anomalies in time series and graph data. The new extension of RSF is also benchmarked against state-of-the-art anomaly detection algorithms.

The contributions of this thesis are as follows:

1. **Streaming model:** The extension of RSF for the streaming model using reservoir sampling. The sliding-window model is also considered, where the goal is to detect anomalies for the most recent data, as old data often becomes outdated in practical applications.

2. **Distributed model:** The extension of RSF for the MapReduce model, which is a de facto model for the distributed setting. In this model, data is distributed among multiple machines and the goal is to perform anomaly detection on all the data using low communication cost. This is achieved through early anomaly selection and a size-reducing sketch method for RSF.

3. **Distributed streaming:** The extension of RSF for the distributed streaming model is also investigated. This model is a generalisation of the previous two models and is explained later.

4. **Benchmarking:** A benchmark of RSF against (Extended) Isolation Forest [33] and Robust Random Cut Forest [31].

5. **Time series data:** The application of RSF for time series data, where shingling is important to detect anomalous patterns in the data. The use of the Autoperiod [66] method is proposed to automatically detect the period for use as the shingle size.

6. **Graph data:** The application of RSF for graph data using the SpotLight [27] method. Here, different subgraphs of an underlying graph are revealed in different time intervals and the goal is to detect those subgraphs that are believed to be anomalous.

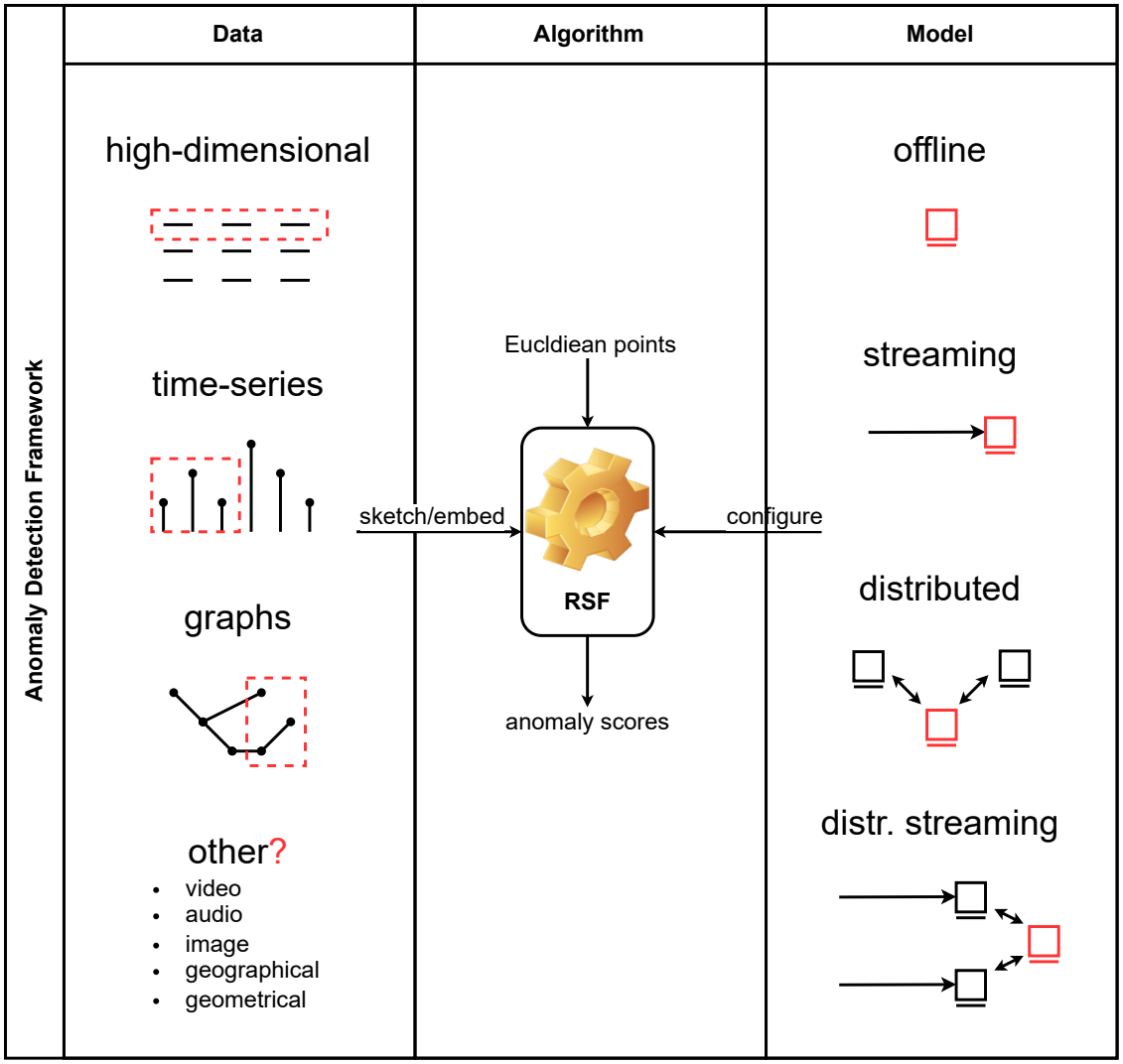


Figure 1.1: An overview of the anomaly detection framework.

An overview of this generic anomaly detection framework built upon RSF is illustrated in figure 1.1. The left column shows different types of input data. RSF is built for data in high-dimensional Euclidean spaces. Special datasets (e.g., graphs or time series) must first be converted (i.e., embedded) into high-dimensional datasets. Several embeddings are discussed in this thesis. The same approach can be applied to other data such as video and audio. Once converted, data is fed into the RSF machinery, which associates anomaly scores with the input points. The right column shows the newly proposed extensions of RSF can also handle input data given in the context of a number of different models. An anomaly detector that can handle all these different scenarios would be very appealing [41, 9].

The rest of this thesis is outlined as follows. Chapter 2 covers the necessary background knowledge used for the algorithms and experiments. Chapter 3 formalises the various versions of RSF and provides some analysis. Chapter 4 contains all the experiments that were performed. Chapter 5 gives a conclusion.

Chapter 2

Background

This chapter covers the necessary background knowledge used for the algorithms and experiments. Section 2.1 covers preliminaries, including the computational models considered and performance metrics used for the experiments. Section 2.2 gives an overview of the field of anomaly detection and briefly explains the state-of-the-art algorithms that are used for benchmarking. Section 2.3 introduces the tools and techniques used for the development of the new extensions of RSF, including Autoperiod [66]; SpotLight [27]; and distributed sampling [17].

2.1 Preliminaries

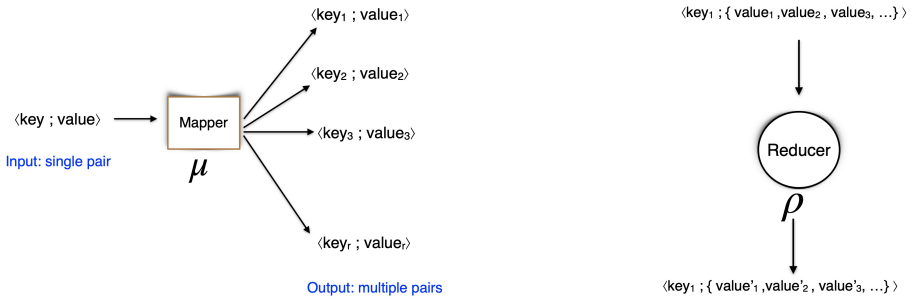
2.1.1 Computational Models

Streaming. In the streaming setting [49], a machine receives many data entries at a high rate. It is assumed a machine does not have enough memory to store all of its input. In this model, entry i with value v_i of the input data arrives in chronological order at a time t_i. There are several variations of the streaming model, including:

- **Insertion-Only model [6]:** The input data entries are revealed one at a time.
- **Sliding-Window model [19]:** Given a stream of the underlying data and a window size W, the goal is to compute and maintain a function for the window of the most recent W elements.
- **Dynamic model [36, 29]:** The stream consists of inserts and deletes of items, where an item can be deleted if it was inserted before.

It further depends on how many passes over the data an algorithm takes. Streaming algorithms are desired to have (poly)logarithmic storage and per-element runtime requirements.

Distributed. In the distributed setting, data and computation are divided over multiple machines. One popular distributed model is MapReduce, formalised by Karloff; Suri; and Vassilvitskii [38]. Here, an input of size n is initially distributed among t machines, each with a local space of size s. Computation takes place in synchronous rounds in which each machine performs local computation on its data and then sends messages to other machines. Input consists of key-value pairs and a round consists of a mapper; shuffle; and reducer phase. In the mapper phase, each key-value pair is mapped to a multiset of key-value pairs. In the shuffle phase, entries with the same key are aggregated and distributed. In the reducer phase, each list of values corresponding to the same key is mapped to a new list of values



(a) Mapper. (b) Reducer.

Figure 2.1: The mapper and reducer operators of the MapReduce model.

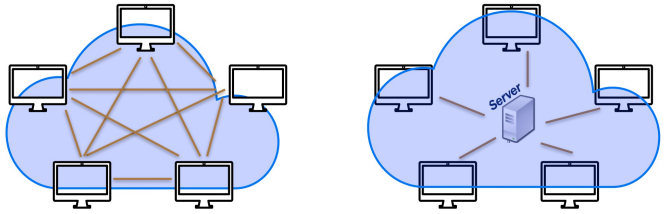


Figure 2.2: The two communication modes of the MapReduce model.

with that key. Figure 2.1 shows a diagram for a mapper and reducer. Many algorithms can be parallelised by interleaving these kinds of parallel/sequential operations. The MapReduce model can be further subdivided into the following models:

- **Peer2Peer:** When communication can occur between any two machines.
- **Coordinator:** When communication has to go through a dedicated machine called the coordinator.

Figure 2.2 shows a diagram of both communication models. Several metrics are used in determining the practicality of an algorithm in this model, including:

- **Communication complexity:** The number of *communication rounds* used to perform a task.
- **Space and time per machine:** The space and running time used by a machine at each stage of an algorithm.
- **Total space:** The sum of the space used per machine.
- **Total time:** The running time across all machines and rounds.

MapReduce algorithms are desired to use sublinear space per-machine (i.e., $o(n)$), sublogarithmic (i.e., $o(log n)$) communication complexity, and polynomial (i.e., $n^{O(1)}$) work and time per machine.

Distributed Streaming The distributed streaming model is introduced by Cormode, Muthukrishan, Yi, and Zhang [18]. It can be seen as a generalisation of the streaming and coordinator models. This model assumes a distributed platform of k sites (or machines) $S_1, \cdots, S_k$ and a coordinator C. Each machine has an input stream and can communicate with a coordinator

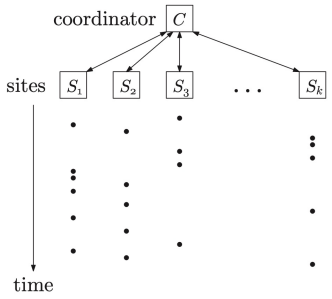


Figure 2.3: An overview of the distributed streaming model. Taken from figure 1 in [18].

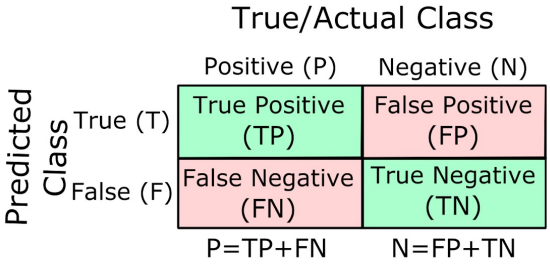


Figure 2.4: The confusion matrix for a binary classifier. Taken from figure 1 in [63].

via one-way or two-way communication. Figure 2.3 shows a diagram of the model. It is easy to see how the previous models can be obtained by varying the number of input machines and space per machine.

2.1.2 Performance metrics

This thesis uses several performance metrics to evaluate the performance of RSF against other anomaly detection algorithms. The base of all these performance metrics is the well-known confusion matrix (see figure 2.4). The matrix consists of the True Positive (TP), False Positive (FP), False Negative (FN), and True Negative (TN) classes.

An anomaly detection algorithm can be seen as a binary classifier. Given a dataset $\mathscr{D}$ of data entries, the goal is to label every entry as $True$ if it is an anomaly or $False$ if it is a normal instance. To evaluate the performance of a classifier, a ground truth labelling of the dataset $\mathscr{D}$ is often provided. Then, in the confusion matrix, the classifier output is the predicted class and the ground truth is the true/actual class. Based on the confusion matrix, define the following basic measures:

- The False Positive Rate (FPR) or false alarm rate, given by $\frac{FP}{FP+TN}$.
- The True Positive Rate (TPR); recall; or hit rate, given by $\frac{TP}{TP+FN}$.
- The precision, given by $\frac{TP}{TP+FP}$.

Next is an explanation of each performance metric used in this thesis, based on the work by Tharwat [63]:

Precision. Given is a dataset $\mathscr{D}$ of size n, of which n_1 entries are labelled as anomalous and $n_1 = n - n_0$ entries are labelled as normal data by a ground truth labelling. The contamination of the dataset is defined as $c = \frac{n_1}{n}$. The F-measure or F-score is defined as the geometric

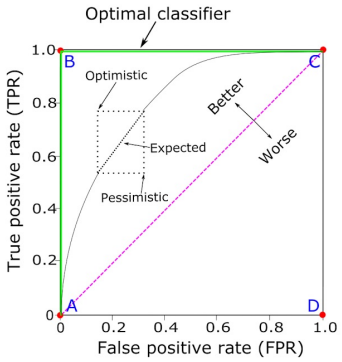


Figure 2.5: A schematic of the ROC curve (continuous). Taken from figure 5 in [63].

mean of the precision (p) and recall (r), and is given by $(p + r)/pr$. The anomaly detectors considered here actually assign continuous anomaly scores to the dataset entries. These are normalised between 0 and 1 and a higher score implies an entry is believed to be more anomalous. To evaluate the F-measure, a discrete class division is required. In this thesis, this is done by marking the $n_1 = cn$ entries with the highest anomaly score as the anomalies and comparing those to the ground truth. As such, the F-measure will be used as an indication of how contaminated an algorithm is given perfect knowledge of the frequency distribution of the anomalous and normal classes. Note that this approach also means each false positive implies a false negative. As a result, the precision and recall (and thus the F-measure) will always be equal. For this reason, just the precision is reported.

ROC(AUC). The receiver operating characteristics (ROC) curve plots the TPR (y-axis) against the FPR (x-axis) (see figure 2.5). Each combined (FPR, TPR) result is a point in the ROC space. This means a single configuration of a binary classifier only constitutes one point. Using the continuous anomaly scores, points can be created at every possible anomaly threshold level. The area under the curve (AUC) is used as a summarising scalar value for the overall performance. This can be calculated by summing the trapezoids underneath the data points, similar to a Riemann sum. The intuition of the ROCAUC score is to capture the degree to which an algorithm can separate the negative and positive classes. It is not sensitive to skewed class distributions, as both the TPR and FPR are ratios. This does make it dangerous to compare the ROCAUC score between different datasets. Note that different ROC curves can have the same AUC value.

PR(AUC). The precision-recall (PR) curve is similar to the ROCAUC curve, but instead plots the precision (y-axis) against the recall (x-axis) (see figure 2.6). Note that both the ROC and PR curve use recall (TPR), but on a different axis. The AUC value can be computed in a similar way as with the ROC curve, resulting in the PRAUC score. In contrast to the FPR used for the ROC curve, the precision can either increase or decrease for different threshold values. In other words, the PR curve can fluctuate, while the ROC curve is concave. Using the precision also makes the PR curve sensitive to skewed class distributions.

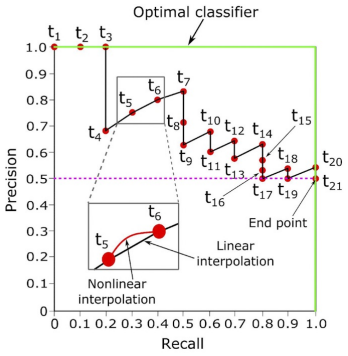


Figure 2.6: A schematic of the PR curve (discrete). Taken from figure 10 in [63].

2.2 Anomaly detection

The amount of information shared digitally grows rapidly, and so does the need to develop ways to cope with this massive data. One of the core concepts in analysing big data is the concept of anomaly detection, also known as outlier detection. In the anomaly detection field, the goal is to seek parts of data that are in some way different from the majority of the data. Indeed, Chandola; Banerjee; and Vipin, in their extremely well-cited survey (p. 15) [16], state:

> Anomaly detection refers to the problem of finding patterns in data that do not conform to expected behavior.

There are many applications of anomaly detection, including:

- Removing outliers from a training dataset before the training phase to obtain more accurate results in machine learning [25].
- Detecting anomalies in network traffic to find data that was sent by an attacking party (intrusion detection) [53].
- Detecting potential cases of credit card fraud [11].
- Finding and removing noise from a dataset before performing analysis on the data to ensure the validity of results [56].

Chandola et al. look at how various research areas, application domains, and problem characteristics feed into the development of anomaly detection algorithms. What follows is a review of the known anomaly detection algorithms, using their extensive coverage of the topic:

Anomaly types. Anomalies can be classified into the following types:

- *Point anomalies* are the simplest type of anomaly, referring to a single point being anomalous with respect to the rest of the data.
- *Contextual anomalies* are only anomalous within a given context. This can be through contextual attributes like rainfall depending on location or through noncontextual attributes like the global average rainfall being made up of single measurements.
- *Collective anomalies* are groups of points that form an anomaly only when considered together.

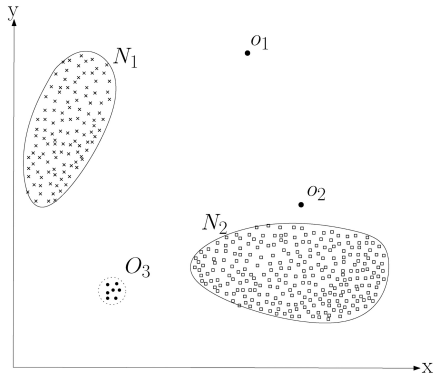


Figure 2.7: Two ground truth clusters (N_1, N_2), two point anomalies (o_1, o_2), and a microcluster of anomalies (O_3). The anomalies are with respect to the two clusters. Taken from figure 1 in [16].

This thesis will mostly consider point anomalies. Collective anomalies will be considered in the context of graphs. Figure 2.7 shows an artificial example of a dataset with point and collective anomalies.

Learning models.

- *Supervised* methods assume the data is split into train and test data. The train data consists of anomalous and normal points, which are labelled in order to learn the appropriate algorithm parameters to differentiate the two classes.
- *Semisupervised* methods use training data which only consists of normal points.
- *Unsupervised* methods do not require any sort of training data.

Clearly, each of these classes is applicable to an increasingly broad extent. This thesis will only be concerned with unsupervised anomaly detection.

Algorithm classes. In general, anomaly detection algorithms can be divided into the following classes:

- **Classification-based:** These are methods that use training data to learn parameters for some type of underlying classification model. Some examples of such models are neural networks [14], Bayesian networks [47], support vector machines [44], and rule-based models [22]. These models can provide good and efficient detection performance, but good training data may not always be available or of limited relevance.
- **Distance-Based:** These are methods that rely on a distance metric between points. One advantage of these types of anomaly detection algorithms is that they are unsupervised and so, there is no need to provide labelled data. The class can be subdivided as follows:
  - **Nearest-Neighbours:** These methods use the distance of a point to its k-th nearest neighbour or k nearest neighbours, or distance within some other local neighbourhood as a measure of the anomalousness of a point. For example, the Local Outlier Factor [12] uses the k-nearest-neighbours to compute a relative density value for a point (using similar concepts as DBSCAN [26]) to avoid the problem of varying densities between different clusters of normal data.

  - **Clustering:** These methods try to cluster the data, leaving the points not belonging to a cluster as anomalies. Well-known clustering algorithms that are used include k-means [48] and DBSCAN [13].

  The two classes share a number of similarities, among which is the assumed existence of local neighbourhoods in the data. The performance also relies on the distance measure used.

- **Statistical-based:** This type of algorithm can be further subdivided into the following:

  - **Parametric:** Techniques in this sub-class model datasets as being sampled from some underlying statistical distribution. Concrete examples are the use of Gaussian [43] or regression [58] models.
  - **Nonparametric:** Techniques in this sub-class do not assume a specific statistical distribution. Concrete examples are histograms [30] and kernel functions [40].

  These categories provide robust methods that can provide statistical guarantees about their results. However, the underlying data distribution and the interplay between various features thereof can prove difficult to describe in practice.

More recently, a survey by Samariya and Thakkar [59] identifies three more classes:

- **Ensemble-based [28]:** This class of methods combine the results of multiple different anomaly detectors and a consensus mechanism to come to a final anomaly labelling. Provided these anomaly detectors do not suffer from the same drawbacks, these methods trade runtime complexity for robustness.
- **Subspace-based:** This class of methods perform anomaly detection on a number of reduced subspaces of the full feature space. One example is to use random Gaussian projection to obtain subspaces and then analyse these [20]. Although reported to be better at finding 'hidden' anomalies, considering many subspaces can involve high computational costs while it may be irrelevant work.
- **Isolation-based:** The isolation-based anomaly detection algorithms are based on isolating anomalies using early cuts from normal data. This will be explained in detail later in this chapter. It has been shown empirically akin to Liu et al [23] that isolation-based algorithms and models that use randomization techniques outperform other types of anomaly detection algorithms such as distance-based [12, 62, 50, 37, 39] and density-based [15, 60, 26] algorithms. Due to this reason, we consider the isolation-based anomaly detection model in this thesis and develop various distributed and streaming algorithms for it. Isolation Forest (iForest) [45] is the defining algorithm of this class. Other variations of this algorithm have been proposed recently, including Extended Isolation Forest (EIF) [33] and Robust Random Cut Forest (RRCF) [31]. These methods have some of the greatest runtime performance but still lack in either accuracy or scalability (or both). This will be explained later in this thesis as well.

Our contribution: The goal of this thesis is to develop an unsupervised isolation-based anomaly detection algorithm that is scalable (i.e., it can be easily implemented in the streaming and distributed models).

From here, section 2.2.1 introduces the iForest algorithm; section 2.2.2 discusses Extended Isolation Forest, which is a generalisation of iForest; and section 2.2.3 covers Robust Random Cut Forest, a novel isolation-based method employed at Amazon. There are many more iForest variants, but these are the state-of-the-art algorithms RSF is benchmarked against here.

2.2.1 Isolation Forest

Isolation Forest (iForest) was introduced by Liu, Ting and Zhou [45]. They mark the main shortcoming of previous anomaly detection methods as focusing on building a profile of normal data (e.g., clustering-based methods) rather than the anomalies. Many of these algorithms involve a high computational cost in building these profiles as well.

The fundamental assumption behind iForest[45, 23] is that anomalies are "few and different". That is,

- **Majority assumption:** The number of *normal* data points far outnumbers the number of *anomalies*.
- **Deviation assumption:** The attribute values of *anomalies* deviate substantially from that of *normal* data.

iForest Construction. Let $P \subset \mathbb{R}^d$ be a point set of n points in a d-dimensional Euclidean space $\mathbb{R}^d$. The iForest algorithm constructs an iForest data structure. The iForest data structure is an ensemble of t iTrees. For every iTree T_i for $i \in \{1, 2, \cdots, t\}$, a subset $X_i \subseteq P$ of size ψ is sampled uniformly at random. Liu, Ting and Zhou [45] empirically show that a good choice for the two parameters t and ψ would be $\psi = 256$ and $t = 100$. They showed that these values are effective across a wide range of problems.

An iTree is built as a recursive binary partition tree on its subsample by repeatedly splitting among a uniformly random dimension at a random value. This random value is generated uniformly at random in the range of the current subset along the split dimension. The splitting continues until only one point remains, being 'isolated', or a depth limit is reached ($\lceil \log_2 \psi \rceil$ by default). An example is shown in figure 2.8.

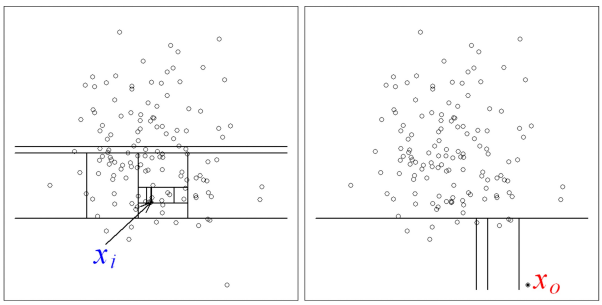


Figure 2.8: An example of splitting until the depth limit is reached (left) and isolating a point early on (right). Taken from figure 1 in [45].

The path length to the leaf that contains a point in an iTree corresponds to the number of splits needed to isolate it. This is used as the measure of how anomalous a point is believed to be; a lower depth being more anomalous (i.e., easier to isolate) and vice versa. When the leaf is at the depth limit, a penalty score is added to compensate for the lack of further splitting. The rest of the tree is modelled as a BST of the leaf's points, and the average search path length of a BST of that size is used as the penalty. Since it is normal data that is supposed to end up lower in the tree, such an approximation is adequate. The main goal is to isolate the anomalies from the normal data, not to differentiate normal data points from each other.

Scoring function for an iTree T_i: Let dep be the depth limit that is set for iTrees in the iForest. Let $x \in P$ be an arbitrary point. Let $\mathscr{A}(x, T_i)$ denote the anomaly score of x for iTree T_i, computed as follows:

- If the leaf ℓ_x in the iTree T_i that contains x is at a level less than dep, then $\mathscr{A}(x, T_i) = Level(\ell_x)$, where $Level(\ell_x)$ is the level of the leaf ℓ_x in the iTree T_i.
- If the leaf ℓ_x in the iTree T_i that contains x is at level dep, then $\mathscr{A}(x, T_i) = dep + c(n)$, where n is the number of points inserted into the leaf, and $c(n) = 2H(n-1) - 2(n-1)/n$.

Here, $H(i)$ is the i-th harmonic number, approximated as $\ln(i) + \gamma$, where $\gamma \approx 0.5772156649$ is the Euler–Mascheroni constant. $c(n)$ is the average path length of unsuccessful search in a BST of size n [54] (as cited by [45]).

iForest scoring. Once an iForest is constructed, it can be used to compute the anomaly score of every point $x \in P$. Every point $x \in P$ is fed into every iTree of the iForest and the average score is computed. This average is compared to the average search path length one would obtain if x was inserted into a 'normal' BST of size ψ. The final score is normalised such that a score of 0 indicates a normal point, and a score of 1 an anomalous point. It should be mentioned that every point $x \in P$ is fed into the iTrees, but not stored. It is only used for scoring. Since the subsample size ψ and number of trees t are small constants, the construction process is efficient and the scoring process is essentially linear in the size of the input. In addition, construction and scoring can be easily parallelised on a per-tree basis.

Scoring function for an iForest F: Let $x \in P$ be an arbitrary point. Let $H(x) = E[\mathscr{A}(x, T_i)] = \sum_{j=1}^{t} \mathscr{A}(x, T_i)/t$ be the average anomaly score of x among all the iTrees of the iForest F. Let $\mathscr{A}(x, F)$ denote the anomaly score of x for iForest F, given by:

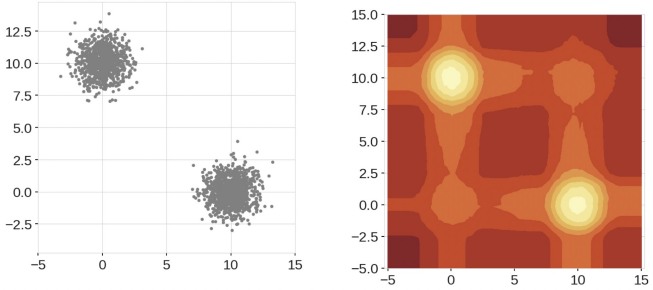
$$\mathscr{A}(x, F) = 2^{-H(x)/c(\psi)}.$$

The sampling that is done for every iTree overcomes two problematic effects in the anomaly detection task:

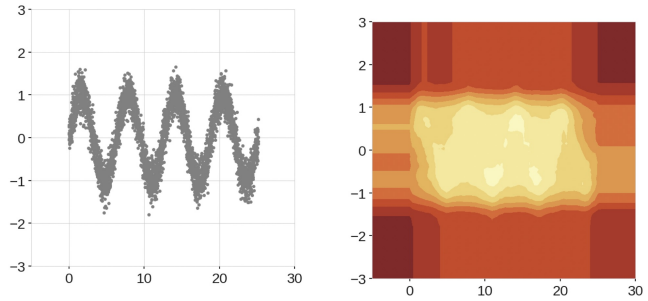
- **Swamping:** The swamping phenomenon refers to detecting a normal point as an anomaly, i.e., a false negative. Normal data may be close to anomalous points, making it harder to differentiate the two. Subsampling avoids sampling normal and anomalous points that are close by.
- **Masking:** Masking refers to detecting an anomaly as a normal point, i.e., a false positive. Two anomalous points may be very close by, making it harder to isolate both. This is again avoided by subsampling.

2.2.2 Extended Isolation Forest

As an improvement on iForest, Hariri; Kind; and Brunner introduced Extended Isolation Forest (EIF) [33]. Through a number of examples, the authors try to highlight some of the problems of iForest and motivate the improvements they incorporated. Two of these examples are shown in figure 2.9. For the first example with the two blobs, two interesting effects can be spotted: axis-aligned rectangular regions of lower anomaly scores and 'ghost' clusters where these regions overlap. The expected pattern is two round regions around where the blobs are, with scores sharply converging further from the two. Similar effects occur for the



(a) Two blobs.



(b) Sine wave.

Figure 2.9: Score maps produced by iForest. Darker indicates more anomalous. Taken from figures 2 and 3 in [33].

sine wave of the second example. Here, the overlapping rectangular regions cause very low scores between the periods of the sine wave, while these are expected to be relatively high. Of course, these are 2D examples that can be visualised well, and one may wonder how this extends into higher dimensions with more complex data.

An explanation of the problems can be found by looking at the splits produced by iForest, as shown in figure 2.10. The bias that iForest produces is grounded in the fact it uses axis-aligned splits. These extend outwards horizontally and vertically. The effect of this is even more striking when looking at the splits of all iTrees of an iForest together. Recall that the anomaly score uses the average of the scores among all iTrees of an iForest.

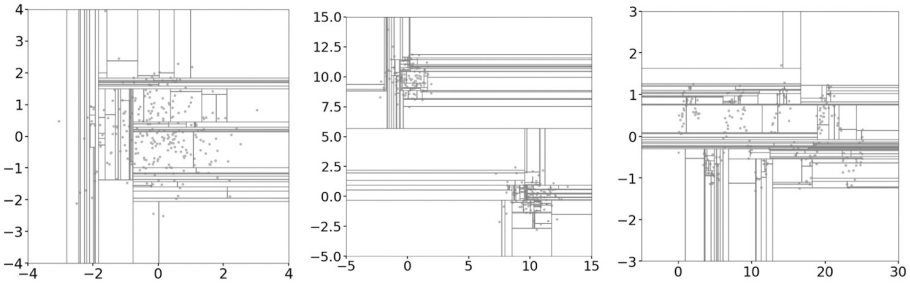
As a remedy to these problems, the authors propose two different improvements:

1. **Input rotation:** Randomly rotating the data of each tree has the effect of randomly rotating the axes along which is split. One disadvantage here is that it is not obvious how this can be done in higher dimensions, though it is possible (e.g., see [10]).

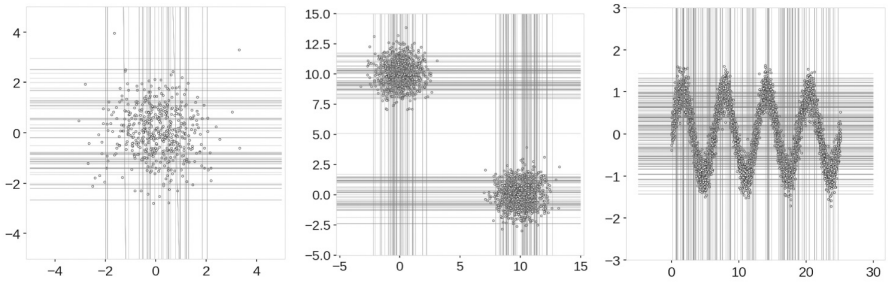
2. **Hyperplane splits:** At each recursive step, the splitting procedure now picks a uniformly random pivot point in the range of the values of the current partition subset. A corresponding direction vector is generated from a uniformly random point on the N-dimensional unit sphere, where N is the dimension of the input. This should not be done naively, as this can lead to a biased sample. Picking each vector component from the standard normal distribution works.

Note that option 1 generalises iForest, which is obtained when using the identity matrix for the rotation. Option 2 in turn generalises option 1, which is obtained when all hyperplanes are aligned with the rotated axes.

Experiments show that both approaches lead to improved performance and less variance between the scores among the trees of a forest. Using hyperplanes gives the best results. Compare the splits of iForest shown in figure 2.10 to those of EIF shown in figure 2.11. The authors note it is also possible to sample the hyperplane slopes among a reduced number



(a) Splits of a single tree.



(b) Distribution of splits among a forest.

Figure 2.10: Splits produced by iForest for three toy problems. Taken from figures 6 and 7 in [33].

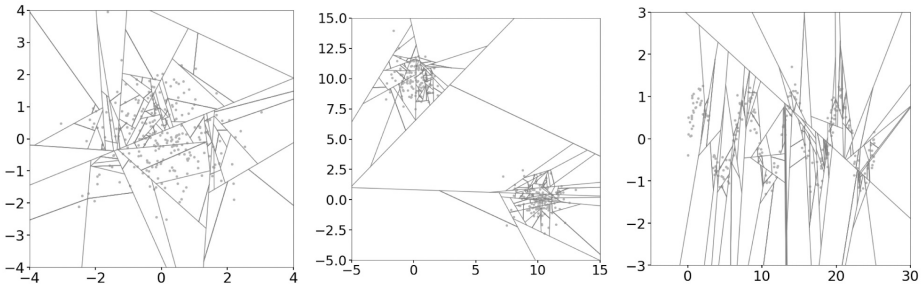
of dimensions of the unit sphere. This is referred to as the extension level of EIF. Using N dimensions (extension level $N-1$) gives the fully extended version of EIF, while extension level 0 (1 dimension) gives back the original iForest algorithm. Experiments show that higher extension levels reduce the bias of iForest and that for high-dimensional data, the fully extended version works best.

2.2.3 Robust Random Cut Forest

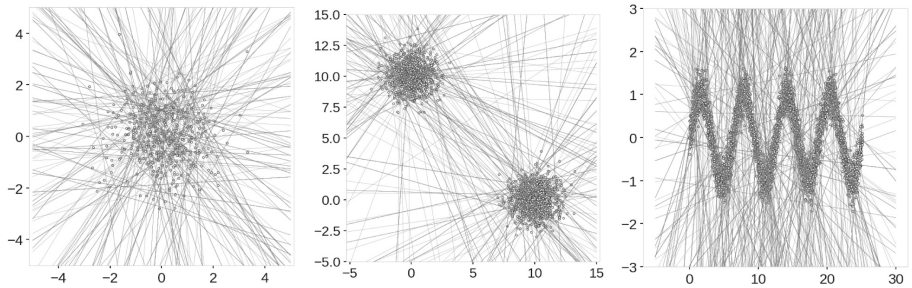
Motivated by the increasingly large ingress of data, Guha; Mishra; Roy; and Schrijvers introduce Robust Random Cut Forest [31]. The goal of their study is twofold: to formalise the definition of an anomaly and to develop an efficient detection algorithm that works for dynamic streaming data. Similar to an iTree, a Robust Random Cut Tree (RRCT) recursively partitions data using random splits. These splits, however, are now weighted according to each dimension's range. With an artificial example, the authors argue that many splits along irrelevant dimensions are performed otherwise. Similar to an iForest, an RRCF consists of an ensemble of independent RRCTs.

Before jumping to the anomaly score used by RRCF, it is useful to state the operations that make it possible for RRCF to work on dynamic streaming data:

- **Insertion:** The insert operation works by walking down from the root of an RRCT. At each node, the bounding box is updated. Using the new bounds, a split is generated. If this split isolates the new point from the rest of the tree, a new parent node is created with the existing tree and the new point as children. Otherwise, the previous split is used and the process repeats at the appropriate child node.
- **Deletion:** The delete operation is simpler. It finds the isolating node of a point and replaces its parent node with its sibling node. Afterwards, the bounding boxes are updated upwards.



(a) Splits of a single tree.



(b) Distribution of splits among a forest.

Figure 2.11: Splits produced by EIF for three toy problems. Taken from figures 9 and 10 in [33].

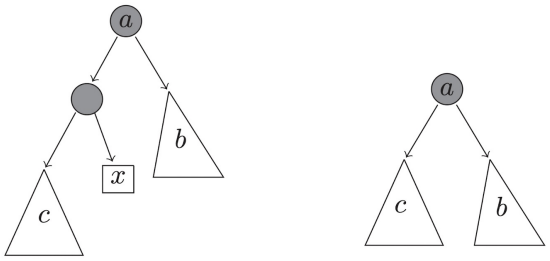


Figure 2.12: Right to left: During insertion, x was isolated from subtree c. Left to right: deletion of x. Taken from figure 1 in [31].

Figure 2.12 shows a schematic of the effect of insertion and deletion. When a subsample is maintained over a stream (e.g., using reservoir sampling), the updates to this subsample can be directly applied to each RRCT using the operations.

RRCF uses the increase in model complexity as a result of insertion as a measure of anomalousness. This works under the assumption that an anomaly itself is easy to describe, but makes it harder to describe the rest of the data. This increase in model complexity is defined as the displacement of a point $Disp(p)$. It turns out their definition of displacement coincides with the number of nodes in the sibling node of the isolating node of a point. What is instead used is a more involved codisplacement $CoDisp(p)$ value. The expected codisplacement (averaged over the RRCTs) is the final anomaly score.

For a real-life application, it is shown how RRCF can be used in a streaming fashion with time series data. RRCF is also shown to outperform iForest on an artificial experiment where the goal is to detect a sudden flat section in a sine signal. Both experiments will come back later.

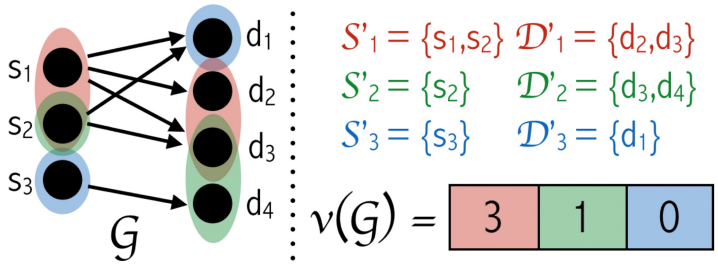


Figure 2.13: Construction of a frequency from randomly matched edge nodes. Taken from figure 3 in [27].

2.3 Extensions

This section introduces the tools and techniques used for the development of the new extensions of RSF, including Autoperiod [66]; SpotLight [27]; and distributed sampling [17].

2.3.1 Graphs and SpotLight

Graph data is different from a stream of numerical data points. Many different algorithms exist to perform anomaly detection on graphs specifically. Eswaran, Guha, and Mishra introduce SpotLight [27]. The difference is that this is not an anomaly detection algorithm on its own but instead enables anomaly detection on graphs using an existing anomaly detection algorithm. The focus here is on weighted directed graphs that change over time. The goal is to detect the sudden appearance of dense subgraphs in near real-time using sublinear memory. An important application here is that of internet traffic. When modelling server requests, attacks such as (D)Dos (sending an overload of requests to a single server) lead to very dense subgraphs as compared to normal and lightweight usage by various individuals.

Density. Consider the input as a stream of graphs $g_1, ..., g_n$, which are each a part of the complete evolving graph G. Each incoming graph consists of a list of edges made up of a source node; destination node; and a positive non-zero weight. The anomalousness of an incoming graph $g_i \subset G$ can be based on the distribution of the densities of its subgraphs. Using the sum of the edge weights of each subgraph as a density measure, this results in a vector of size $2^{|g_i|}$ to be used for anomaly detection. Of course, this is computationally intractable.

SpotLight. The idea of SpotLight is to select a fixed number of K random subgraphs instead. This is done with K independent pairs of hashing functions that select an edge's source and destination node with probabilities p and q, respectively. If both are selected, the node weight is added to the sum corresponding to that pair of hashing functions. The resulting K-dimensional density vector is a sketch for the incoming graph. An example is shown in figure 2.13. The intuition is that each hash function pair is a spotlight highlighting a random subgraph of the full graph. A sudden appearance of a dense cluster of edges is expected to partially overlap with some of these subgraphs, and thus be reflected in the sketched density vector. Algorithm 1 provides pseudocode for the SpotLight algorithm.

Properties. To strengthen the latter claim, the authors provide a theoretical analysis and explain why this approach works well in tandem with an anomaly detection algorithm. Define the SL-distance as the expected squared Euclidean distance between two SpotLight

Algorithm 1 SpotLight

Require: rank decomposition k
Require: source hash probability p
Require: target hash probability q
  procedure init
    $h^p \leftarrow$ list of k functions hashing independently to 0 with probability p
    $h^q \leftarrow$ list of k functions hashing independently to 0 with probability q
  procedure next(graph G)
    $W \leftarrow$ 0-vector of size k
    for each edge e in graph G **do**
      for $1 \leq i \leq k$ **do**
        if $h_i^p(e.\text{source}) = 0$ and $h_i^q(e.\text{target}) = 0$ **then**
          $W_i \leftarrow W_i + e.\text{weight}$
    return W

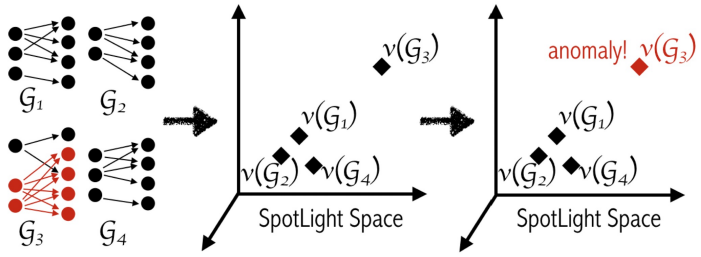


Figure 2.14: An overview of SpotLight's steps. Larger and more local additions (simulating anomalies) lead to a larger distance between sketches, which can then be detected as anomalies. Taken from figure 2 in [27].

sketches. Two such SL-sketches are ϵ-SL-far if their distance to the sketch of some other graph differs by at least ϵ. Two properties are worked out, for which parameter settings and their theoretical guarantees are provided:

1. **Focus-Awareness:** The smaller the region of a graph to which unit edges are added uniformly at random, the larger its SL-distance to the original graph. As a consequence, the appearance of star graphs leads to a higher SL-distance as compared to the appearance of a matching graph. These two types of graphs are used as representatives of a focused attack and normal, more widespread communication.

2. **Anomaly detection criterion:** Consider two graphs obtained from a given original, one where n edges are randomly added in a small $n \times n$ region and one where edges are added throughout. The algorithm computes sketches that are ϵ-SL-far with respect to the original with high probability.

A final overview of the steps, including the use of an anomaly detector at the end, is shown in figure 2.14. In some experiments on real-world data, SpotLight is shown to outperform existing prior baseline detectors and provide interpretable and interesting results. The experiments use RRCF as the anomaly detection algorithm. An extension of one of these experiments will be performed in section 4.5.

2.3.2 Shingling and Autoperiod

Shingling. Shingling is a technique where an input stream of points is changed to an input stream of shingles, which are groups of consecutive points viewed as one single point. It is equivalent to using a small sliding window, which emits a new point (resulting from concatenating its contents) after each shift. As such, shingling can be seen as another sketching technique, providing a summary of local data. Some pseudocode to do this efficiently using a priority queue is given in algorithm 2. Shingling can be used to take into account the context of data, albeit much smaller than the context that a sliding window or (time-sensitive) sampling provides. The anomaly detection problem moves from being about individual points to one about recognising small patterns.

Algorithm 2 Shingling

```
Require: single size s
  procedure init
    Q ← DoubleEndedQueue()                          ▷ back to front order
  procedure next(point p)
    if size(Q) < s then
      pushFront(Q, p)
    else
      popBack(Q)
      pushFront(Q, p)
      return contents of Q as a single point
```

Time series. Shingling is an important tool to enable anomaly detection in time series data, as the same values can occur at later times but in a different local (anomalous) pattern. Think for example of a stable sine function that has a temporary frequency spike, while maintaining the same amplitude. As suggested in [31], a good rule of thumb for the shingle size is to use the 'natural' periodicity of a signal. To automatically detect such periodicity is no trivial matter. Assuming an input stream can assume only a fixed set of values, fingerprint sketching can be used to do it in one pass exactly with high probability and sublinear memory [24]. This method also allows for efficient computation of the 'distance' of a signal to a given period. Real data, however, is often continuous and noisy, calling for a different approach.

Autoperiod. One such approach is the Autoperiod method introduced by Vlachos, Yu, and Castelli [66]. They mention how the use of periodicity is prevalent in many types of fields like natural science, medicine, and transport-related industries. An overview of the method is shown in figure 2.15. At the basis of their method is the Fourier transform, which represents a signal as a linear combination of complex sinusoids. The Fourier transform can be used to obtain two types of estimators of the Power Spectral Density (PSD) of a signal, which is a distribution of the frequencies (and by extension, periods) occurring in the signal. The estimators are:

1. **Periodogram:** The periodogram is obtained from the squared lengths of the Fourier coefficients. It is easily interpretable but works on bins that get more coarse for higher frequencies. It also suffers from 'spectral leakage', where frequencies that are not multiples of the bin widths influence the whole spectrum.
2. **Autocorrelation:** The circular autocorrelation function (ACF) is a convolution between shifted versions of the signal and can be computed as a dot product in the frequency domain (the Fourier transform). It is harder to interpret, as peaks occur at multiples of the frequency as well. High frequencies of low amplitude also weigh less.

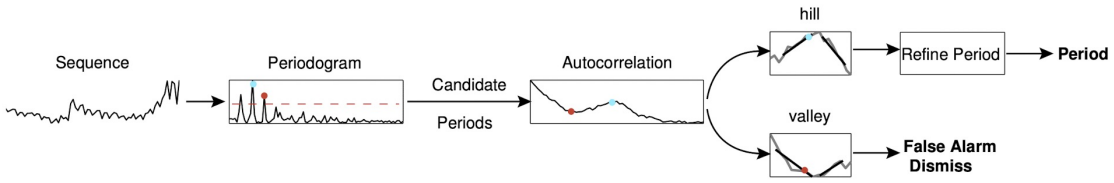


Figure 2.15: A schematic of Autoperiod. Taken from figure 3 in [66].

It is interesting to note that the ACF is the inverse Fourier transform of the periodogram, making them duals. The idea of Autoperiod is to use the strengths of both. In the first place, the periodogram is used to find candidate frequencies. The peaks are filtered out by determining a threshold from several signal permutations that retain power levels but lose periodicity information. Then the corresponding frequencies are checked to be a valley (false alarm) or hill in the ACF. If it is a hill, the value of the hilltop is determined and used instead to resolve the bin granularity problem. To do this reliably, linear regression is used on local segments around the candidates.

The Fourier transform can actually be updated in streaming fashion using the incremental Momentary Fourier Transform (MFT) [51] (as cited by [66]). The authors note that this makes it possible to translate this method to the streaming setting as well, with both a growing and sliding window version of the MFT. The study further features some real-world examples to showcase the effectiveness of the method. Recently, an improved version of Autoperiod was published [55]. This method, called CFD-Autoperiod, includes some extra steps to resolve issues for signals that are noisy or have multiple periodicities. However, since there was no readily available implementation, the original Autoperiod method is used for this study.

2.3.3 Distributed Sampling

Like iForest, RSF uses subsamples of the input to build its underlying tree data structures. Chung, Tirthapura, and Woodruff [17] propose a message-optimal algorithm to perform uniform random sampling (without replacement) in the distributed streaming model. Pseudocode for the machines and coordinator is provided in algorithms 3 and 4, respectively.

Let s be the sample size. The idea is to associate each incoming point with a random weight. Each machine keeps a possibly outdated copy (u_m) of the s-th minimum weight of the sampled points at the coordinator (u_c). When the weight of an incoming is below the s-th minimum weight, both the points and its weight are sent to the coordinator to update the sample. If applicable, the coordinator sends back the new largest sample weight (only to the machine received from).

Algorithm 3 DistributedSamplerMachine

```
procedure init
    u_m ← 1.0
procedure update(weight u_c)
    u_m ← u_c
procedure next(point p)
    w ← randNumber(0.0, 1.0)
    if w < u_m then
        emit((p, w))                                  ▷ to coordinator
```

Algorithm 4 DistributedSamplerCoordinator

Require: sample size s
  procedure init
    $u_c \leftarrow 1.0$
    $Q \leftarrow$ PriorityQueue() ▷ key: weight, value: point
  procedure update((point p, weight u_m))
    $u_m \leftarrow u_c$
    insert(Q, u_m, p)
    if size(Q) $> s$ **then**
      popMax(Q)
      $u_c \leftarrow$ maxKey(Q)
      emit(u_c) ▷ to machine received from

Chapter 3

Algorithm

This section develops the Random Shift Forest (RSF) algorithm. First, section 3.1 presents the offline algorithm in terms of its operators and configuration parameters. Next, section 3.2 shows how these operators can be used to adapt RSF for the streaming setting in various ways. Finally, section 3.3 discusses two ways in which the offline algorithm can be distributed among multiple machines.

3.1 Offline RSF

For a large part, RSF follows similar ideas as those presented for Isolation Forest (iForest) in section 2.2.1. The main difference is to use a deterministic splitting procedure and introduce randomness through a translation of the input points. Rather than defining the algorithm as a recursive partitioning of the input, it is defined in terms of point-wise operations. This makes the construction and maintenance of RSF much more flexible while retaining the same time and space complexity.

Random Shift Tree. The Random Shift Tree (RST) is the underlying tree data structure used by RSF. Like an Isolation Tree (iTree), it is a partition tree with points stored in the leaf nodes. Unlike an iTree, all internal nodes at the same level use the same split dimension and always split their respective bounding box halfway along this dimension.

Let B be the axis-aligned bounding box (exactly) containing all of the input. The bounds of the input are thus assumed to be known. Another possibility is to normalise all the input and use the unit cube. Let B' be the bounding box obtained by adding the range of B to its upper bound along each axis. Each tree applies a random shift to its points, uniformly sampled from the range of B along each axis. The resulting points are guaranteed to be contained by bounding box B'. Figure 3.1 shows how B is obtained from B', and how points are randomly shifted.

Let d be the dimension of the input. Each tree fixes an order L of uniformly sampled dimensions among which to split halfway at a given tree depth. A tree is initialised with a maximum node capacity and maximum depth. Initially, it has an empty root node at depth 1 (level 0) with a bounding box B'. Figure 3.2 shows two examples of trees with the same configuration but a different shift. Notice how the splits are similar but occur at different places.

A tree supports the following operations:

- **Insert:** Find the leaf node with the bounding box containing input point p. If the leaf node has not reached its maximum capacity, insert the point into the node. If it is full, split the node's bounding box among the split dimension corresponding to its depth and distribute its points among the two new child nodes accordingly. Repeat

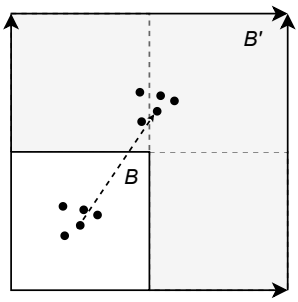
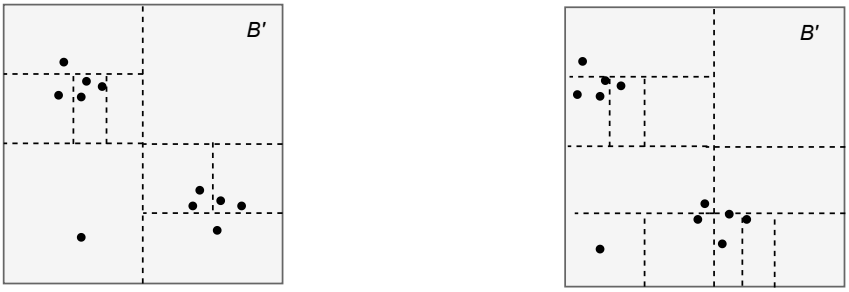


Figure 3.1: B' is obtained from B by adding its range to the upper bounds along every axis. Points in B are randomly shifted to anywhere within B'.



(a) Tree 1. (b) Tree 1.

Figure 3.2: Two trees with the same input and split order but different shifts.

the procedure at the next child node. If a node is at the maximum depth, always insert the point. The complete pseudocode is given in algorithm 5.

- **Delete:** The delete operation is the direct inverse of the insert operation. First, find the leaf which contains the point p to be deleted, and remove it from its list of points. Next, to undo any of the invalid node splits that may now be present, traverse the tree upwards from the leaf's parent to the root. If a node has only leaves as children and their total number of points does not exceed the maximum capacity, the children are deleted and the parent becomes a new leaf with their combined points. Note that it is not necessary to consider any other nodes in the tree, as an internal node is guaranteed to have more points in its subtree than the node capacity, since it is split and no other points are removed. The complete pseudocode is given in algorithm 6.
- **Score:** Scoring works the same as for iForest. An anomalous point is interpreted as one that is easier to isolate, i.e., is located higher up the tree. First, find the leaf with the bounding box containing input point p. The basic anomaly score is the path length to this node. Since the tree uses a maximum depth, an extra penalty is added to leaves at this level, but only if they exceed the maximum node capacity. Let $c(n)$ again denote the average path length of an unsuccessful search in a BST of size n. Evaluate this value for the number of points of the node as the additional penalty. It serves as an indication of the additional path length if the node were further expanded as a BST. The complete pseudocode is given in algorithm 7.

Example. Figure 3.3 shows example output of the splits of a tree for a number of 2D toy datasets. The datasets are random subsamples of size 1024 of those introduced in section 4.2. The trees are built on a random subsample of 128 points and use a node capacity

Algorithm 5 Insert

Require: RandomShiftTree T
Require: point p
  $p_s \leftarrow p + T$.shift
  $n \leftarrow T$.root
  while not isLeaf(n) **do** ▷ find leaf
    $n \leftarrow c$ in n.children where contains(c.boundingBox, p_s)
  while true **do** ▷ split as necessary
    if n.depth $< T$.maxDepth and size(n.points) = T.maxPoints **then**
      split(n, T.splits[n.depth])
      $n \leftarrow c$ in n.children where contains(c.boundingBox, p_s)
    else
      insert(n, p_s)
      return

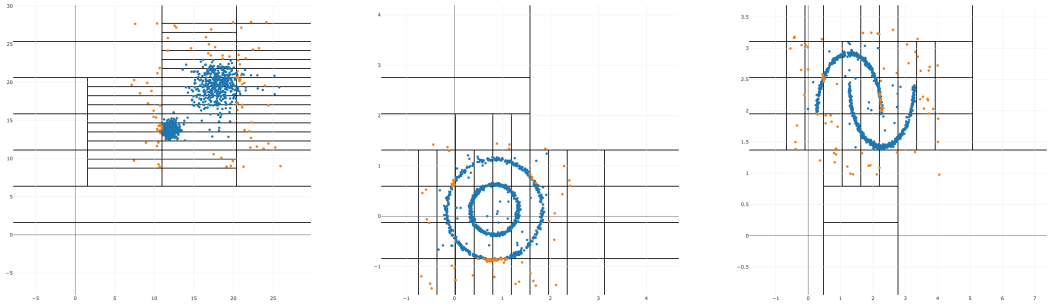
Algorithm 6 Delete

Require: RandomShiftTree T
Require: point p
  $p_s \leftarrow p + T$.shift
  $n \leftarrow T$.root
  while not isLeaf(n) **do** ▷ find leaf
    $n \leftarrow c$ in n.children where contains(c.boundingBox, p_s)
  delete(n, p_s)
  while hasParent(n) **do** ▷ contract as possible
    $n \leftarrow n$.parent
    onlyLeaves $\leftarrow$ isLeaf(c) for all c in n.children
    sizeSum $\leftarrow$ sum of size(c.points) for all c in n.children
    if onlyLeaves and sizeSum $< T$.maxPoints **then**
      pointSum $\leftarrow$ union of c.points for all c in n.children ▷ can contain duplicates
      n.points $\leftarrow$ pointSum
      clear(n.children)
    else
      return

Algorithm 7 Score

Require: RandomShiftTree T
Require: point p
  $p_s \leftarrow p + T$.shift
  $n \leftarrow T$.root
  while not isLeaf(n) **do** ▷ find leaf
    $n \leftarrow c$ in n.children where contains(c.boundingBox, p_s)
  penalty $\leftarrow$ 0
  if n.depth = T.maxDepth and size(n.points) > T.maxPoints **then**
    penalty $\leftarrow$ c(size(n.points))
  pathLength = n.depth - 1
  return pathLength + penalty

of 2.



(a) Blobs. (b) Circles. (c) Moons.

Figure 3.3: Example output on 2D toy datasets for RSTs configured subsample size 128 and node capacity 2.

Random Shift Forest. Finally, a Random Shift Forest (RSF) is just an ensemble of a fixed number of RSTs. The following parameters are used:

- **Number of points:** Also referred to as the sample size, the number of points n each RST is expected to contain also dictates the maximum depth of a tree. Like with iForest, it is simply set to $\lceil \log_2(n) \rceil$.
- **Number of trees:** The idea of using multiple RSTs with random splits and shifts is to avoid the bias from splitting deterministically, mimicking the random split values of iForest. The final anomaly score of a point is the average of its scores among all RSTs. Like is done for iForest, it will be useful to normalise this score to a value between 0 and 1. Given an average score s, the reported anomaly score follows as $2^{-s/c(n)}$, where c is again the average path length of an unsuccessful BST search and n the sample size. As such, a score of 0 means a point is very unlikely to be anomalous, and a score of 1 means a point is very likely to be anomalous.
- **Granularity:** The maximum node capacity is determined by a granularity parameter g. It works in such a way that the first $\lfloor k/g \rfloor$ RSTs are set to a max of 1, the second $\lfloor k/g \rfloor$ RSTs to a max of 2, and so forth till the last $\lfloor k/g \rfloor$ RSTs with a max of g. A higher max means that for some trees it takes more points to split a node, thus leading to increased anomaly scores. This can aid in detecting microclusters of anomalies and avoid the masking effect. Microclusters of anomalous points that are very close would otherwise end up far down the tree with a low anomaly score.

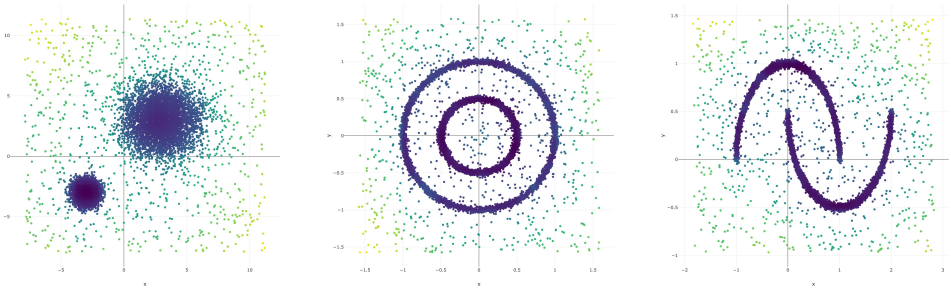
The insertion and deletion operations on an RSF are simply repeated for each of its RSTs.

Complexity. Consider a subsample size n and maximum node capacity w. The maximum tree height is set to $h = \lceil \log_2 n \rceil$. For tree insertion, in the worst case, a node needs to be split at each level, where at most w points need to be redistributed among the two new child nodes. Such a case always occurs when the w points each have the same coordinates and thus will never be split. The resulting runtime complexity of insertion is $\mathcal{O}(w \log_2 n)$. Regardless of the node capacity, however, a point is always involved in one node split at each level. Thus, the amortised runtime complexity is $\mathcal{O}(\log_2 n)$. For tree deletion, a similar argument applies. In the worst case, a parent node needs to be contracted at each level, where at most w points are collected from its two child nodes. Again, a point can only be involved in the

contraction of a node once at each level. This also gives a runtime complexity of $\mathcal{O}(w \log_2 n)$ in general and $\mathcal{O}(\log_2 n)$ amortised. For tree scoring, in the worst case, a leaf node is situated at the maximum depth. There are no further operations involving each point of a node, giving a runtime complexity of $\mathcal{O}(\log_2 n)$ in general. There are at most $\mathcal{O}(2^h) = \mathcal{O}(2^{\log_2 n}) = \mathcal{O}(n)$ leaf nodes per tree. Together with the internal nodes and the n points themselves, this gives a space complexity of $\mathcal{O}(n)$. An implementation may further reduce space by using a shared point store. Note that the influence of the dimension on the point size and bounding box is ignored. There is no further overhead in maintaining a forest of k trees, such that only an extra factor k applies to all operations and storage, where the granularity g of a forest determines the worst-case regarding the node capacity (when $w = g$).

RSQF. Visser [65] also proposed a special variant of RSF that can be more effective in 2D. This variant's 2D bounding box is always split among both its axes. This means each internal node has four child nodes and there are no random split dimensions. The resulting data structure is called a Random Shift Quad Tree (RSQT) as part of a Random Shift Quad Forest (RSQF). Of course, it is not hard to imagine similar variants for higher dimensions, but this quickly becomes intractable. RSQF is not further considered in this thesis, but can otherwise be extended in the same way as RSF.

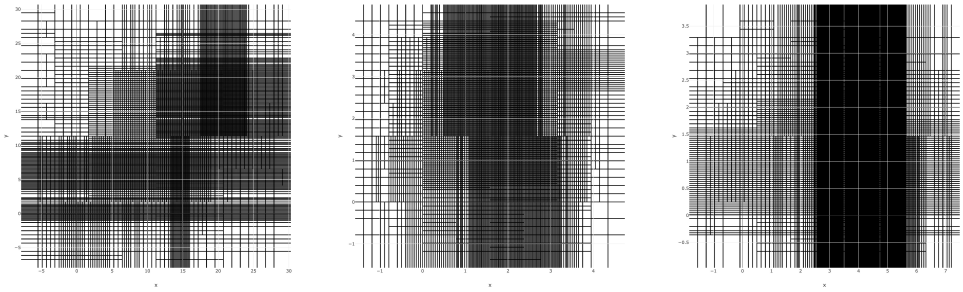
Example. Refer back to the RST output shown in figure 3.3. Figures 3.4 till 3.6 show the normalised point scores, tree splits, and heatmaps for RSF on the full versions of the same three toy datasets. RSF is configured for 64 trees of size 1024, using granularity 4. The relatively large sample size is the tipping point after which RSF can differentiate the regions between the moon and circle objects. The distribution of the splits may be hard to interpret since each tree splits for a differently shifted input.



(a) Blobs. (b) Circles. (c) Moons.

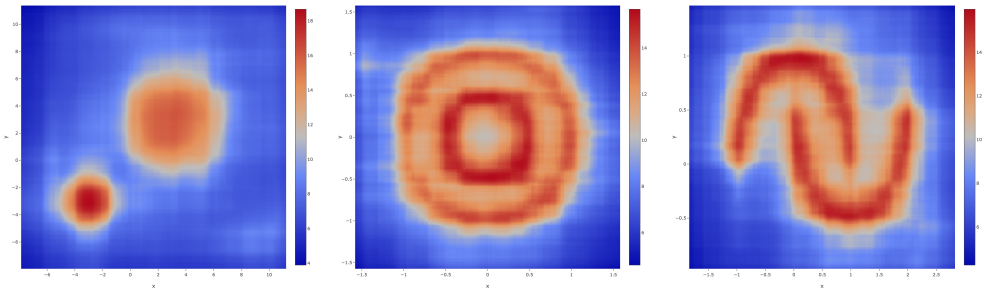
Figure 3.4: Normalised point scores for RSF on three toy datasets.

Bias. Refer back to the discussion of Extended Isolation Forest (EIF) of section 2.2.2. It was shown that the splitting procedure of Isolation Forest (iForest) leads to unwarranted regions of increased anomaly scores. Figure 3.7 shows the output of iForest, EIF, and RSF on a similar dataset of two blobs. The dataset is generated using the `make_blobs` function of the Python package scikit-learn [52]. The two normal blobs of total size 10000 were generated from two Gaussian distributions centred at $(0, 10)$ and $(10, 0)$, both with a standard deviation 2. Anomalies are added as two Gaussian distributions of total size 50 centred at $(0, 0)$ and $(10, 10)$, both with standard deviation 2. The algorithms are configured with 32 trees of size 256. Like EIF, in contrast to iForest, RSF's splitting procedure avoids the unwanted side-effect of ghost clusters of higher anomaly scores. The problem with iForest is that splits of different trees tend to be focused around the same normal data and extend into the same



(a) Blobs. (b) Circles. (c) Moons.

Figure 3.5: Splits of RSF trees on three toy datasets.



(a) Blobs. (b) Circles. (c) Moons.

Figure 3.6: Heatmap of anomaly scores for RSF on three toy datasets.

direction. However, since RSF applies a random shift to the data, splits may be anywhere within a range of the whole bounding box. Along every dimension, with respect to the first split, all data may be either completely below or above the splitting value. This example also shows that this impairs iForest's ability to correctly detect anomalies. Compare these results to figure 3.6 of the example above as well. Some bias of the axis-aligned splits still seems to remain. Note that these results are based on just one random run. The experiments in section 4.2 will further benchmark the performance difference.

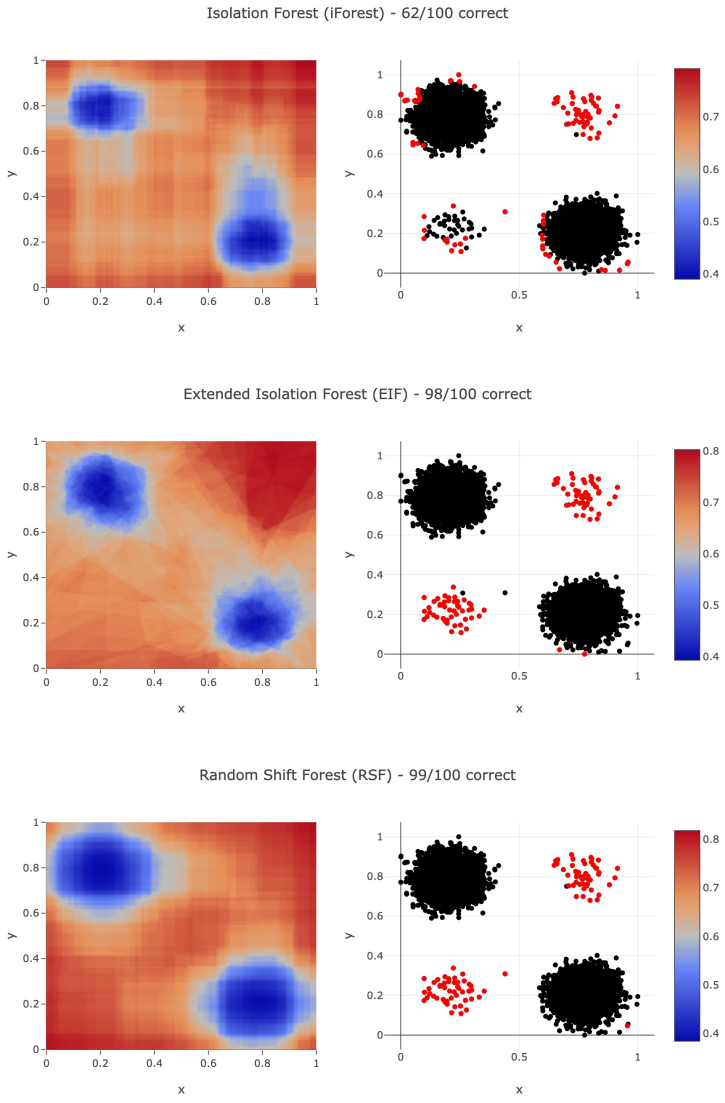


Figure 3.7: Heatmap of the anomaly scores and anomalies detected by iForest, EIF, and RSF on a dataset of two normal and two anomalous point clusters of Gaussian noise.

3.2 Streaming RSF

A subsample of the input data is used to build the RSF trees that are used for scoring. This subset can be incrementally obtained and updated by sampling from a stream. What remains is to keep RSF up to date with this sample. This section proposes three ways to do this using the presented RSF operators, covering both the insertion-only and sliding-window models. The dynamic streaming model is not considered.

Split sampling. Split sampling is the most straightforward sampling technique. The term sampling is not really appropriate (but used for consistent naming anyway) since it is actually deterministic. First, in the insertion phase, the first n incoming points are inserted into an RSF, where n is the configured tree size. From there, in the scoring phase, all other incoming points are scored using the built RSF. This is analogous to the train-test partitioning of data, except that no random shuffling is applied. This is most useful when there is some sort of ground truth data representative of all the normal data and mostly free of anomalies. When anomalies occur in the insertion phase, it increases the anomaly score of similar anomalies, making them harder to differentiate. The main drawback is that this approach assumes the distribution of the data is static. For example, when a bus schedule is changed, the number of trips at the new times is expected to spike, but this should not repeatedly be detected as an anomaly. In addition, the old bus times should be 'forgotten' to be normal data.

Algorithm 8 Split sampling

```
Require: tree size n
  procedure init
      F ← RandomShiftForest()
  procedure next(point, index)
      if index < n then
          insert(F, point)
      else
          s ← score(F, point)
          return s
```

Reservoir sampling. To mitigate the issue of clean initial data, one idea is to keep a running uniform sample of the data. The effectiveness of taking a uniform sample was already proven for the offline version of the algorithm implemented by Visser [65]. This task can be performed using reservoir sampling. There are multiple correct ways to do this, and the one presented here is just one. The most important thing is that a running sample is maintained of the stream so far, instead of a sample that is only completed at the end of the stream. After all, a stream may theoretically never and, and points are supposed to be scored based on the input history so far (they cannot all be stored). Sampling can be done for each tree separately or for the whole RSF together. Pseudocode to do the latter is given in algorithm 9.

Window sampling. Finally, to remedy the issue of changing distributions, a sliding window approach may be applied. Here, the goal is to keep a running sample of only the last w elements. The approach is to use hash functions that hash a point to 0 with a probability of n/w. This means that in a window of w elements, there is an expected number of n points that are hashed to 0. These are inserted into the tree. The points in the current window are stored. When a point is replaced by a new one, it can again be hashed to check if it was inserted. If so, it is removed. Again, this can be done for each tree separately (using k hash

Algorithm 9 Reservoir sampling

```
Require: tree size n
  procedure init
      F ← RandomShiftForest()
      L ← List()                                   ▷ 0-based index
  procedure next(point, index)                     ▷ 1-based index
      if size(L) < n then
          push(L, point)
          insert(F, point)
      else
          s ← score(F, point)
          j ← randInt([0, index));
          if j < n then
              remove(F, L_j)
              L_j ← point
              insert(F, point)
          return s
```

functions) or for the whole RSF together. Note that in the former case, the points still only need to be stored once. Pseudocode to do the latter is given in algorithm 10.

Complexity. Consider RSF with k trees of size n and a stream of length $s >> n$. Take the granularity to be a negligible small constant. For each of the streaming methods, it holds that each point is only ever inserted, deleted, or scored once (per tree). For all practical values of k, n, s, the runtime and space requirement of RSF are logarithmic in the input size. Thus, combined with the logarithmic complexity of the operations, it follows that the proposed methods make up an efficient streaming algorithm. Split sampling requires no additional space outside of the RSF. Reservoir sampling takes up $\mathcal{O}(n)$ extra space for each forest or tree to maintain the reservoir. This could be improved by using a shared point store. For window size w, window sampling takes up $\mathcal{O}(w)$ extra space to maintain the points in the current window (ignoring the hash functions). This can also be improved by only storing the points that are inserted (in at least one tree). Also assuming $s >> w$, both do not incur significant overhead for practical values and thus maintain the streaming algorithm efficiency.

Sine wave. Guha et al. [31] benchmark Robust Random Cut Forest (RRCF) against Isolation Forest (iForest) using an artificial sine wave signal. This example showcases the various streaming RSF versions on an adaptation of this signal and compares sampling per tree (split) or for the forest as a whole (joint). The sine wave is given amplitude 50, period 100, and vertical translation 100. It is sampled at each integer value along the x-axis. An anomaly of 25 units is placed at a multiple of 100, where the wave is set to a fixed value of 80. This experiment uses 1000 points with the anomaly placed at 700. This example also adds Gaussian noise with mean 0 and standard deviation 5. The RSF parameters are set to 40 trees of size 256. A shingle size of 4 is also used since the change point would not be detected otherwise. This is because the flat signal is still within the normal range of the wave bounds. It is the combination of the flat value jumping back to the normal wave value that constitutes the anomaly. The experiments of section 4.4 elaborate on this topic.

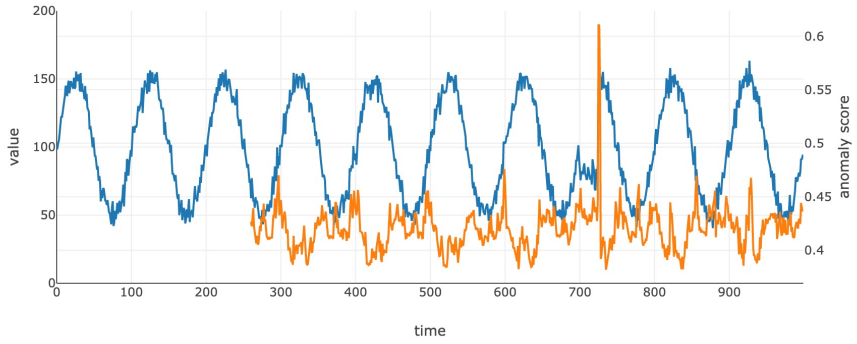
Figure 3.8 show the results for joint and split sampling using the reservoir and window sampling (size 512) methods. Split sampling is also included. RSF seems to produce comparable anomaly scores, regardless of whether split sampling or split/joint reservoir/window or sampling is used. There are some subtle differences due to the difference in methods. Split sampling seems to show a bit more variability in its scores, and window sampling has

Algorithm 10 Window sampling

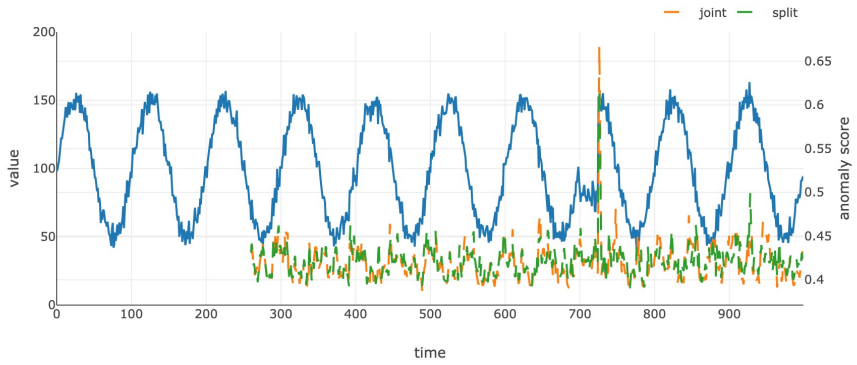
Require: tree size n
Require: window size w
  procedure init
    $F \leftarrow$ RandomShiftForest()
    $Q \leftarrow$ PriorityQueue() ▷ back to front order
    $h \leftarrow$ function hashing to 0 with probability $\frac{n}{w}$
  procedure handleOld
    (point, index) $\leftarrow$ popBack(Q)
    if h(index) $= 0$ **then**
      remove(F, point)
  procedure handleNew(point, index)
    pushFront(Q, (point, index))
    if h(index) $= 0$ **then**
      insert(F, point)
  procedure next(point, index)
    if size(Q) $< w$ **then**
      handleNew(point, index)
    else
      $s \leftarrow$ score(F, point)
      handleOld()
      handleNew(point, index)
      return s

some additional smaller peaks. The difference may be explained by the fact that reservoir sampling is always based on the whole history of the stream so far, thus taking more noise into account. By default, experiments in this thesis will use sampling per tree individually, as this should avoid masking and swamping effects from affecting a whole forest rather than one tree.

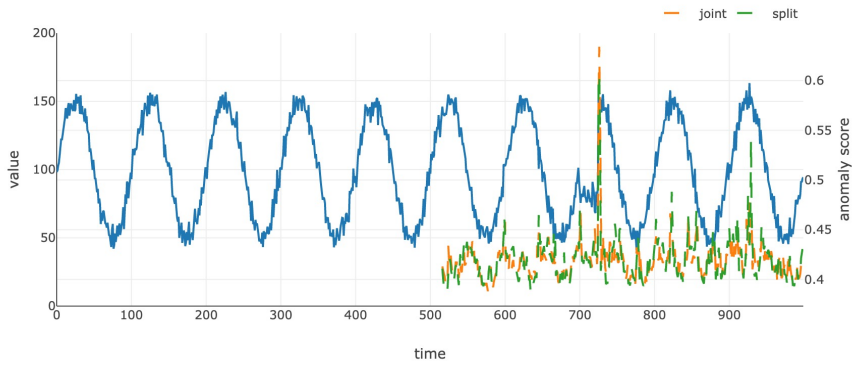
Figure 3.9 shows the results of using split window sampling for shingle sizes 1, 4, and 16. Using a larger shingle size appears to have a slight dampening effect on the anomaly scores. The anomalous event also leads to a widened peak, since more shingled points involve the anomalous jump between values. This may seem disadvantageous, except that this example also shows that the shingle size of 1 is too small and does not lead to the desired results. In contrast to the other shingle sizes, there is no peak marking the end of the flat line.



(a) Split sampling.



(b) Reservoir sampling.



(c) Window sampling.

Figure 3.8: Joint vs. split sampling.

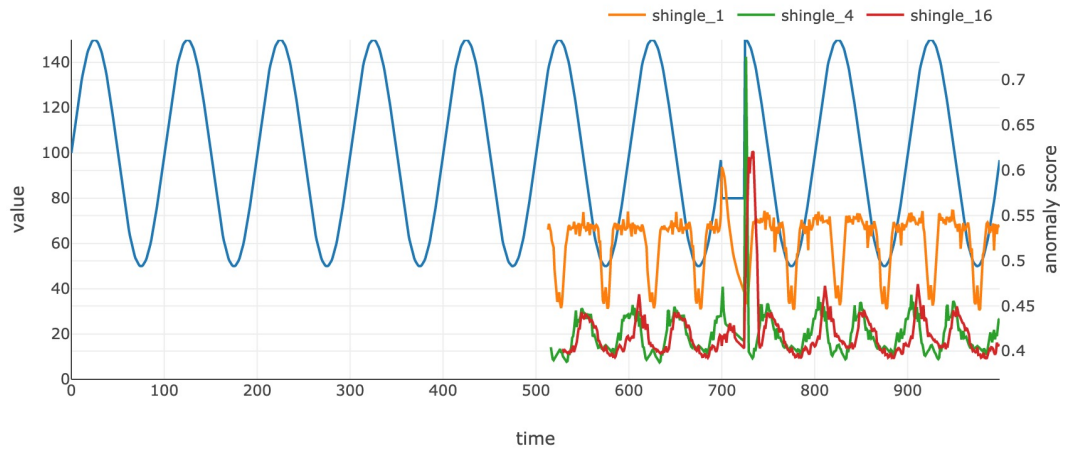


Figure 3.9: Split window sampling with shingle sizes 1, 4, and 16.

3.3 Distributed RSF

Because RSF has a deterministic splitting procedure, it can be fully translated to the distributed setting. It is useful to generalise the notion of nodes to include sketches. Each node corresponds to a subspace of the bounding box, and an inserted/deleted point always follows the same path of nodes down a tree. Now, imagine each node contains a sketch that keeps some information on the points it has seen. When two RSFs with the same configuration but different input are combined, their sketches at each level are compatible. However, for algorithms that use random splits and split values, this may not be the case. Visser [65] formalised a similar argument where RSF is applied to a dynamic stream and nodes use sparse recovery sketches.

In what follows, a simple sketching approach is proposed to reduce the communication overhead of distributed RSF. Afterwards, two version of RSF for two different distributed models will be introduced. The performance of these techniques and methods will be considered in the experiments of seciton 4.2. Note that for this experimental evaluation (i.e., to evaluate the precision), these algorithms use the true number of anomalies n_1. In practice, this can of course be adapted (e.g., by using a threshold based on a tree depth or standard deviation).

Sketching. To reduce the communication complexity of distributed RSF, a simple sketch can be applied. The points in each leaf of each tree are reduced to a set of at most s weighted points. This is done by simply taking the first s points and cyclically increasing the weights of those points by the weights of the excess points. Pseudocode for the approach taken here is given in algorithm 11. To support weighted point sets, only minor adjustments to the tree operations are necessary. The insert operation tries to find a point and increase its weight and otherwise inserts it as a new point with a weight of one. The delete operation tries to find a point and decrease its weight by one, fully deleting it if its weight reaches zero. Node capacity is now interpreted as the total point weight it may contain.

Assuming a tree size of n with maximum depth $h = \lceil log_2 n \rceil$, the runtime complexity follows as $\mathcal{O}(log_2 n)$ since each point can only be removed once. This ignores maintaining correct weights. The weights could also just be ignored until the sketching operation is performed. Equal points need not even be grouped. Assuming a leaf has c points, another

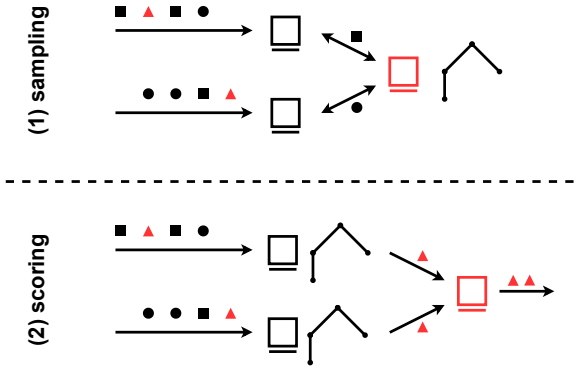


Figure 3.10: A diagram of TwoWayDistrStreams.

possibility is to sample s points and give each a weight of $\lceil c/s \rceil$.

Algorithm 11 Sketch

Require: Weighted point set L ▷ 1-based index
Require: Sketch size s
  numOfPoints ← size(L)
  numOfExcess ← numOfPoints - s
  if numOfExcess > 0 **then**
    for $j \in \{s+1, ..., \text{numOfPoints}\}$ **do**
      $i \leftarrow j$ mod s
      $L[i]$.weight ← $L[i]$.weight + $L[j]$.weight
  truncate(L, s) ▷ keep first s weighted points

TwoWayDistrStreams. This way of distributing RSF is implemented in the distributed streaming model, with two-way communication and two passes over the input streams. A diagram of the approach is shown in figure 3.10. Pseudocode is given in algorithm 12. Given is a randomly or adversarially distributed set of m input streams. As the first pass over the input, sample ψ points in distributed fashion using the method presented in section 2.3.3. Construct an RSF using the sample, sketch it using algorithm 11, and distribute it to all the machines. As the second pass over the input, at each machine, score all the points and send the n_1 most anomalous candidate points to the coordinator. At the coordinator, select the n_1 most anomalous among all the candidates and report those as the final anomalies.

OneWayCoordinator. This way of distributing RSF is implemented in the coordinator model, with one-way communication and one pass over the input. A diagram of the approach is shown in figure 3.11. Pseudocode is given in algorithm 13. Given is a randomly or adversarially distributed input on a set of m machines. Let ψ the desired sample size. At each machine, construct an RSF by sampling ψ/m points for each tree independently. Configure each RSF as if it were for a sample of size ψ. Each tree on each machine with the same index uses the same shift and splits (same random seed). Score the input points using these partial RSFs and select the n_1 most anomalous points. Sketch the partial RSFs using algorithm 11. Send sketched partial RSFs along with the most anomalous points of each machine to the coordinator. At the coordinator, construct one full RSF by merging the trees with matching

Algorithm 12 TwoWayDistrStreams

procedure pass1
Obtain sample Z of size n using algorithms 3 and 4.
On coordinator C, sketch RSF(Z) and send it to all the machines.
procedure pass2
On every machine M_i**:**
Score the incoming stream using RSF(Z).
Send the n_1 most anomalous points to the coordinator.
On coordinator C**:**
Report the incoming points as the anomalies.

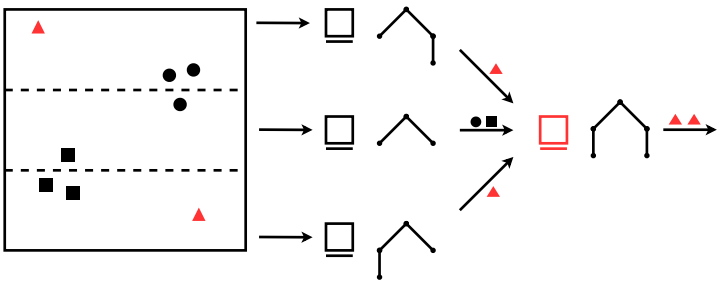


Figure 3.11: A diagram of OneWayCoordinator.

index bottom-up in pairs, inserting points of the one into the other. Use the full RSF to again score the received candidate points. Select the n_1 most anomalous points using these scores and report those as the final anomalies.

Algorithm 13 OneWayCoordinator

procedure round1
On every machine M_i**:**
construct RSF F_i by randomly sampling m/n points from X_i for each tree.
Score the points in X_i using RSF F_i.
Let Y_i be the n_1 most anomalous points.
Sketch RSF(Z_i) to obtain S_i using algorithm 11.
Send S_i and Y_i to the coordinator.
procedure round2
On coordinator C**:**
Let $Y = \bigcup_{i=1}^{m} Y_i$ be the set of points received.
Merge the sketches receives to obtain RSF $S = \bigcup_{i=1}^{m} S_i$.
Score the points in Y using S.
Report the n_1 most anomalous points.

Complexity. Each machine performs offline RSF in some way on its local input. As shown in section 3.1, this maintains the sublinear memory and polynomial runtime requirements for distributed algorithms, as well as the sublogarithmic communication complexity. Although the sketching operation does not give any further guarantees about the latter, it will be empirically shown to make a significant difference in the experiments of section 4.3.

Chapter 4

Experiments

4.1 Experimental setup

Experiments were carried out on a Mac Mini (M1, 2020) with 128GB storage and 8GB memory, running MacOS Monterey. All performance measures are averaged over 32 repetitions and reported as 'mean ± standard deviation'.

RSF. All source code for this thesis is available at GitHub [35]. This includes a Rust implementation of Random Shift Forest (RSF), Python code to download and generate the input data, and Rust code to perform the experiments.

RRCF. The implementation used for Robust Random Cut Forest (RRCF) is the Rust port by Amazon Web Services (AWS) available at GitHub [7].

EIF. The implementation used for Extended Isolation Forest (EIF) is a modified version of the Rust port by Mandery. Both are available at GitHub [46] [34]. The modified version resolves an issue that occurs for datasets with a bounding box with a range of zero along some dimensions. EIF computes the bounds along each dimension and then uses these to randomly sample pivot points for its splits. When the algorithm tries to sample from an open range with equal bounds (which is empty), it crashes. In this case, the modified algorithm instead returns the lower bounds. It is also important to note this algorithm is offline, though it could be modified to be online. Since it does not support the use of a predetermined bounding box, all experiments using EIF use normalised data, such that the testing data used for scoring is not out of bounds with respect to the training data used to construct the trees.

4.2 Classification

This experiment and some of those that follow make use of several toy and real datasets, which are elaborated on below. Table 4.2 provides some general statistics of the datasets. Figure 4.1 shows how the anomalies of the datasets are distributed.

Toy datasets. The toy datasets are generated using the Python package scikit-learn [52]. Each consists of 10,000 normal and 1000 anomalous points, randomly shuffled. The anomalous points are sampled uniformly at random from the bounding box centred at the bounding box of the normal data and scaled by a factor of 1.5. The datasets include:

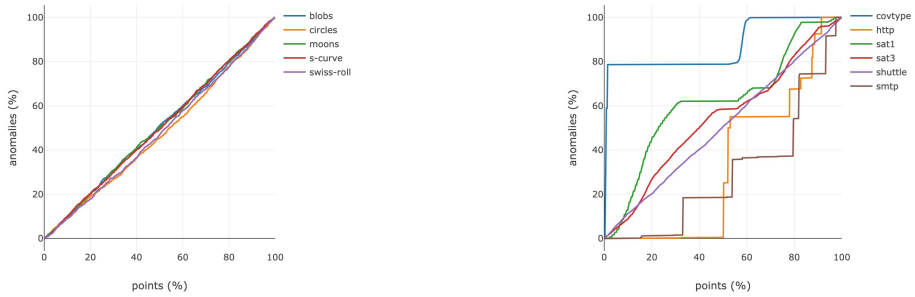
name	points	dimensions	anomalies	contamination
blobs	11000	2	1000	9.1%
circles	11000	2	1000	9.1%
moons	11000	2	1000	9.1%
s-curve	11000	3	1000	9.1%
swiss-roll	11000	3	1000	9.1%

Table 4.1: Toy datasets.

name	points	dimensions	anomalies	contamination
covtype	286048	54	2747	1.0%
http	623091	38	4045	0.6%
sat1	6435	36	703	10.9%
sat3	6435	36	2036	31.6%
shuttle	57990	9	3501	6.0%
smtp	96554	38	1183	1.2%

(a) Real datasets.

Table 4.2: General dataset statistics.



(a) Toy datasets.

(b) Real datasets.

Figure 4.1: Dataset anomaly distributions.

1. **Blobs:** Generated using the `make_blobs` function, using centres $(-3.0, 3.0)$ and $(3.0, 3.0)$ with standard deviations 0.5 and $.5$, respectively.
2. **Circles:** Generated using the `make_circles` function, using factor 0.5 and noise 0.02.
3. **Moons:** Generated using the `make_moons` function, using noise 0.02.
4. **S-Curve:** Generated using the `make_s_curve` function, using noise 0.05.
5. **Swiss-Roll:** Generated using the `make_swiss_roll` function, using noise 0.05.

Real datasets. The real datasets are obtained from the UCI Machine Learning Repository [21]. The datasets include:

1. **Covtype:** Acquired from the Covertype dataset [2]. It consists of cartographic measurements of 30m x 30m cells of several wilderness areas and their corresponding forest cover types. Label 2 is the normal class. Label 4 is the anomalous class.
2. **Http:** Acquired from the KDD Cup 1999 dataset [3] entries with 'protocol_type' of 'http'. It is based on the 1998 DARPA Intrusion Detection Evaluation Program dataset [42]

and contains features of internet connections with labels stating if the connection is normal or the type of attack (an anomaly).

3. **Sat1:** Acquired from the Statlog (Landsat Satellite) dataset [4]. It consists of measurements of 3 x 3 pixel cells of satellite images and the corresponding soil type of the central pixel. Labels 1, 3, 4, 5, 6, and 7 are the normal class. Label 2 is the anomalous class.
4. **Sat3:** Like sat1, but with 3 labels for the anomalous class. Labels 1, 3, 6, and 7 are the normal class. Labels 2, 4, and 5 are the anomalous class.
5. **Shuttle:** Acquired from the Statlog (Shuttle) dataset [5]. It consists of a number of numerical attributes of spacecraft flight. Labels 1 and 4 are the normal class. Labels 2, 3, 5, and 7 are the anomalous class.
6. **Smtp:** Like http, but now using entries with 'protocol_type' of 'smtp'.

Experiment. Benchmark RSF against iForest (EIF with extension level 0); fully-extended EIF; and RRCF. All algorithms are configured for 32 of size 256. RSF uses granularity 4. All data is normalised, as is required for EIF. Each method uses the split sampling approach of algorithm 8. The resulting ROCAUC, PRAUC, and precision metrics based on the scored points are reported. For comparison, RSF using reservoir and (size 1000) window sampling is also included. Both use sampling for each tree independently. Table 4.3 shows the results for the toy datasets and table 4.4 shows the results for the real datasets.

Discussion. In general, in terms of all performance measures, RSF and RRCF are quite close, iForest and EIF scoring similar or slightly worse. Though this might be expected since each algorithm is isolation-based to some extent, it is still interesting to see that each of the algorithms clearly scores best on one or two datasets. iForest and EIF also have an occasional bad score, but it is unclear whether this is due to the underlying method itself or implementation issues. For the toy datasets, the standard deviation of the scores is very low, while it varies a bit for each algorithm when it comes to the real datasets.

Compared to the toy datasets, the real datasets show a greater variability of how well the algorithms can detect anomalies. It seems this can be partially explained by the number of anomalies at the beginning of the dataset, more anomalies being harder and vice versa. This highlights the importance of clean training data.

For the toy datasets, using RSF with reservoir or window sampling does not make a significant difference compared to the split variant of RSF. For the real datasets, especially window sampling shows degraded performance on some datasets. This might be because masking and swamping effects are enhanced in a shorter window, especially when anomalies appear close to each other. For reservoir sampling, there is probably no substantial distribution change when iterating through the data that might give it the upper hand. Overall, it can be concluded that RSF is on par with the state-of-the-art, but the effectiveness of sampling is limited.

algorithm	rocauc	prauc	precision
iForest-split	.96 ± .00	.87 ± .01	.79 ± .01
EIF-split	.97 ± .00	.88 ± .01	.81 ± .01
RRCF-split	.96 ± .00	.88 ± .00	.80 ± .01
RSF-split	.97 ± .00	.89 ± .01	.81 ± .01
RSF-reservoir	.97 ± .00	.89 ± .00	.81 ± .01
RSF-window	.97 ± .00	.89 ± .00	.81 ± .01

(a) Blobs.

algorithm	rocauc	prauc	precision
iForest-split	.83 ± .00	.66 ± .01	.59 ± .01
EIF-split	.84 ± .01	.66 ± .02	.59 ± .02
RRCF-split	.87 ± .00	.70 ± .01	.62 ± .01
RSF-split	.84 ± .00	.67 ± .00	.59 ± .01
RSF-reservoir	.84 ± .01	.68 ± .00	.61 ± .01
RSF-window	.83 ± .01	.66 ± .00	.59 ± .01

(b) Circles.

algorithm	rocauc	prauc	precision
iForest-split	.88 ± .01	.75 ± .02	.68 ± .03
EIF-split	.90 ± .01	.78 ± .02	.71 ± .03
RRCF-split	.94 ± .01	.85 ± .02	.76 ± .02
RSF-split	.91 ± .01	.82 ± .01	.75 ± .02
RSF-reservoir	.92 ± .01	.82 ± .01	.74 ± .01
RSF-window	.91 ± .00	.81 ± .01	.73 ± .01

(c) Moons.

algorithm	rocauc	prauc	precision
iForest-split	.89 ± .01	.75 ± .02	.69 ± .02
EIF-split	.87 ± .01	.68 ± .03	.62 ± .03
RRCF-split	.92 ± .01	.81 ± .01	.74 ± .01
RSF-split	.90 ± .01	.80 ± .01	.73 ± .02
RSF-reservoir	.90 ± .01	.80 ± .01	.74 ± .01
RSF-window	.90 ± .01	.80 ± .01	.74 ± .01

(d) S-Curve.

algorithm	rocauc	prauc	precision
iForest-split	.86 ± .01	.71 ± .02	.64 ± .02
EIF-split	.85 ± .00	.67 ± .02	.61 ± .02
RRCF-split	.89 ± .00	.77 ± .01	.70 ± .01
RSF-split	.87 ± .01	.74 ± .02	.68 ± .02
RSF-reservoir	.87 ± .01	.76 ± .01	.70 ± .01
RSF-window	.87 ± .00	.76 ± .01	.70 ± .02

(e) Swiss-Roll.

Table 4.3: Anomaly detectors benchmark results for the toy datasets.

algorithm	rocauc	prauc	precision
iForest-split	.79 ± .10	.06 ± .08	.07 ± .10
EIF-split	.74 ± .09	.03 ± .04	.03 ± .07
RRCF-split	.95 ± .01	.14 ± .05	.17 ± .06
RSF-split	.85 ± .13	.15 ± .17	.17 ± .19
RSF-reservoir	.64 ± .05	.03 ± .02	.07 ± .06
RSF-window.	.78 ± .05	.09 ± .03	.17 ± .05

(a) Covtype.

algorithm	rocauc	prauc	precision
iForest-split	.97 ± .02	.54 ± .14	.50 ± .12
EIF-split	.90 ± .04	.12 ± .13	.14 ± .16
RRCF-split	.95 ± .00	.47 ± .00	.44 ± .00
RSF-split	.94 ± .01	.44 ± .05	.43 ± .02
RSF-reservoir	.96 ± .01	.39 ± .11	.41 ± .05
RSF-window.	.92 ± .01	.15 ± .02	.20 ± .03

(b) Http.

algorithm	rocauc	prauc	precision
iForest-split	.98 ± .00	.95 ± .01	.89 ± .02
EIF-split	.99 ± .01	.95 ± .04	.89 ± .05
RRCF-split	.98 ± .01	.89 ± .04	.85 ± .03
RSF-split	.98 ± .00	.94 ± .02	.89 ± .02
RSF-reservoir	.96 ± .01	.70 ± .09	.69 ± .06
RSF-window.	.94 ± .01	.61 ± .07	.57 ± .06

(c) Sat1.

algorithm	rocauc	prauc	precision
iForest-split	.69 ± .01	.66 ± .02	.54 ± .02
EIF-split	.68 ± .02	.64 ± .03	.51 ± .04
RRCF-split	.66 ± .01	.64 ± .01	.50 ± .02
RSF-split	.69 ± .02	.66 ± .02	.55 ± .03
RSF-reservoir	.70 ± .02	.64 ± .03	.56 ± .03
RSF-window.	.69 ± .02	.57 ± .04	.51 ± .03

(d) Sat3.

algorithm	rocauc	prauc	precision
iForest-split	.99 ± .00	.89 ± .05	.85 ± .06
EIF-split	.98 ± .00	.89 ± .03	.86 ± .04
RRCF-split	.99 ± .00	.83 ± .05	.79 ± .07
RSF-split	.99 ± .00	.94 ± .02	.94 ± .02
RSF-reservoir	.99 ± .00	.94 ± .02	.94 ± .04
RSF-window.	.98 ± .01	.93 ± .06	.92 ± .07

(e) Shuttle.

algorithm	rocauc	prauc	precision
iForest-split	1.00 ± .00	.94 ± .05	.93 ± .05
EIF-split	.99 ± .01	.54 ± .22	.54 ± .25
RRCF-split	1.00 ± .00	.99 ± .00	.97 ± .00
RSF-split	1.00 ± .00	.98 ± .01	.96 ± .02
RSF-reservoir	1.00 ± .00	.99 ± .00	.96 ± .02
RSF-window.	.99 ± .01	.50 ± .08	.45 ± .08

(f) Smtp.

Table 4.4: Anomaly detectors benchmark results for the real datasets.

4.3 Distributed

Section 3.3 discusses how RSF can be generalised to the distributed setting. This section compares the two versions of distributed RSF that were proposed and compares them to each other and to the offline version of RSF. This includes TwoWayDistrStreams defined in algorithm 12 and OneWayCoordinator defined in algorithm 13.

Experiment. Investigate the influence of varying the number of machines, sketch size, and sample size on the performance of distributed RSF. By default, RSF is configured for 32 trees of size 256, using granularity 4 and 16 machines. The number of machines is varied as 1, 4, and 8. The sketch size is varied as 1, 2, 4, and 8. The sample size is varied as 128, 256, 512, 1024, and 2048.

The toy and real datasets of the classification benchmark experiment of section 4.2 serve as the input. Given the number of machines m, the input is distributed by generating $m-1$ splitting indices uniformly at random in the full list of data points. Note that this can generate any data partitioning possible, from completely balanced to completely balanced.

Next to the precision, each evaluation will report the average tree size (in terms of number of weighted points) of the sketched full RSF (after sampling for TwoWayDistrStreams; after merging for OneWayCoordinator). The results are shown in tables 4.5 and 4.6, and figures 4.2 till 4.9. For comparison, the figures also include an offline baseline, which assumes all data is on one machine; takes random samples for each tree; and applies no sketching.

Discussion. As evident from the tables, varying the number of machines and sketch size does not seem to have any significant influence on the performance of distributed RSF. However, using fewer machines and a smaller sketch size both drastically decrease the average tree size. This can be explained by fewer machines receiving more points, which (being mostly the kind of normal data) leads to more points at the same leaf nodes being sketched.

As evident from the figures, varying the sketch size when varying the sample size does not have a significant influence on the performance of distributed RSF either. In addition, the performance of distributed RSF matches that of offline RSF. Varying the sample size itself does have a varying influence on the performance. For the toy datasets, this is in all cases positive. For the real datasets, it varies, being somewhat positive or negative. This difference may be explained by the absence of the masking and swamping effects, occurring to a lesser extent for the artificial uniform noise of the toy datasets. Also here it can be seen that the sketch size has a dramatic effect on the average tree size. The savings are especially pronounced for real datasets. Real data seems naturally more clustered in comparison to the more stretched shapes of the toy datasets.

Note that these conclusions apply to both algorithms simultaneously. The performance of both is very similar across all datasets. This shows that both are equally valid ways to distribute RSF for their respective domains. Only the average tree size of OneWayCoordinator is somewhat higher. This difference may be due to TwoWayDistrStreams sketching a fully built RSF, while OneWayCoordinator sketches partial trees and then merges these. Again, the more points that are inserted together, the more end up being sketched together.

m	s	avg. tree size	precision
1	1	22.12 ± 1.59	.81 ± .01
1	2	38.32 ± 3.91	.81 ± .02
1	4	60.60 ± 4.09	.81 ± .01
1	8	92.78 ± 4.33	.81 ± .02
8	1	21.84 ± 1.93	.81 ± .01
8	2	37.30 ± 3.29	.81 ± .01
8	4	61.50 ± 3.93	.81 ± .01
8	8	92.74 ± 5.35	.81 ± .01
16	1	22.18 ± 1.91	.81 ± .01
16	2	37.52 ± 2.94	.80 ± .02
16	4	60.18 ± 4.50	.81 ± .01
16	8	92.33 ± 4.89	.81 ± .01

(a) Blobs.

m	s	avg. tree size	precision
1	1	23.54 ± 1.24	.73 ± .02
1	2	44.42 ± 2.82	.73 ± .02
1	4	76.57 ± 3.74	.73 ± .02
1	8	129.23 ± 5.17	.73 ± .02
8	1	24.56 ± 1.14	.73 ± .02
8	2	43.34 ± 2.74	.73 ± .02
8	4	75.61 ± 3.79	.73 ± .03
8	8	128.30 ± 4.56	.73 ± .02
16	1	24.01 ± 1.53	.73 ± .02
16	2	43.82 ± 2.26	.73 ± .02
16	4	75.74 ± 3.50	.73 ± .02
16	8	130.78 ± 6.79	.73 ± .02

(b) S-Curve.

m	s	avg. tree size	precision
1	1	5.26 ± 0.74	.48 ± .03
1	2	9.02 ± 1.03	.49 ± .06
1	4	15.78 ± 1.13	.50 ± .09
1	8	26.61 ± 2.03	.49 ± .05
8	1	5.40 ± 0.62	.49 ± .07
8	2	9.09 ± 0.97	.49 ± .04
8	4	16.10 ± 1.34	.48 ± .04
8	8	26.92 ± 1.54	.48 ± .04
16	1	5.54 ± 0.59	.49 ± .07
16	2	9.78 ± 0.96	.50 ± .04
16	4	15.95 ± 1.72	.50 ± .06
16	8	26.86 ± 2.38	.49 ± .04

(c) Http.

m	s	avg. tree size	precision
1	1	3.06 ± 0.39	.92 ± .08
1	2	5.91 ± 0.59	.91 ± .11
1	4	11.44 ± 0.99	.94 ± .05
1	8	20.68 ± 1.60	.94 ± .05
8	1	3.09 ± 0.40	.93 ± .05
8	2	5.80 ± 0.45	.94 ± .03
8	4	11.30 ± 0.94	.91 ± .11
8	8	20.26 ± 1.23	.93 ± .08
16	1	2.96 ± 0.30	.94 ± .03
16	2	5.92 ± 0.45	.91 ± .09
16	4	10.91 ± 0.94	.93 ± .07
16	8	20.17 ± 1.29	.91 ± .12

(d) Shuttle.

Table 4.5: TwoWayDistrStreams results for varying number of machines m and sketch size s.

m	s	avg. tree size	precision
1	1	22.21 ± 0.60	.81 ± .01
1	2	38.74 ± 1.39	.82 ± .01
1	4	61.43 ± 2.03	.81 ± .01
1	8	92.93 ± 3.67	.82 ± .01
8	1	74.62 ± 2.80	.81 ± .01
8	2	123.06 ± 4.58	.82 ± .01
8	4	183.27 ± 7.08	.81 ± .01
8	8	228.09 ± 9.97	.81 ± .01
16	1	102.81 ± 3.48	.81 ± .01
16	2	167.20 ± 4.42	.81 ± .01
16	4	225.72 ± 4.66	.81 ± .01
16	8	252.22 ± 3.78	.81 ± .01

(a) Blobs.

m	s	avg. tree size	precision
1	1	23.93 ± 1.26	.75 ± .01
1	2	43.98 ± 2.46	.74 ± .02
1	4	77.30 ± 4.02	.75 ± .01
1	8	129.38 ± 5.28	.74 ± .01
8	1	97.80 ± 3.34	.74 ± .02
8	2	166.38 ± 5.31	.74 ± .02
8	4	232.96 ± 5.37	.74 ± .01
8	8	253.08 ± 9.94	.74 ± .01
16	1	125.24 ± 3.67	.73 ± .02
16	2	199.67 ± 5.60	.74 ± .01
16	4	249.64 ± 4.70	.74 ± .01
16	8	252.40 ± 6.03	.74 ± .02

(b) S-Curve.

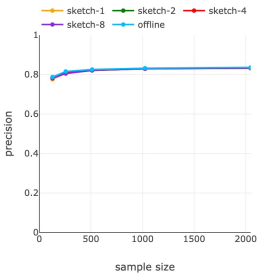
m	s	avg. tree size	precision
1	1	5.22 ± 0.35	.49 ± .04
1	2	9.45 ± 0.82	.48 ± .04
1	4	16.11 ± 1.15	.48 ± .04
1	8	27.73 ± 1.20	.48 ± .04
8	1	19.54 ± 1.49	.49 ± .07
8	2	34.29 ± 2.84	.47 ± .02
8	4	60.91 ± 5.49	.47 ± .02
8	8	106.37 ± 8.41	.47 ± .03
16	1	31.99 ± 2.40	.48 ± .04
16	2	58.03 ± 3.47	.48 ± .05
16	4	102.46 ± 6.51	.47 ± .03
16	8	169.62 ± 6.50	.47 ± .01

(c) Http.

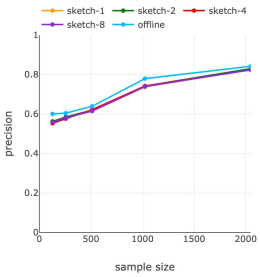
m	s	avg. tree size	precision
1	1	3.00 ± 0.29	.95 ± .02
1	2	5.63 ± 0.43	.94 ± .02
1	4	11.15 ± 0.75	.93 ± .03
1	8	20.20 ± 1.67	.90 ± .09
8	1	19.11 ± 1.51	.94 ± .02
8	2	35.14 ± 2.76	.92 ± .08
8	4	62.74 ± 4.06	.94 ± .03
8	8	106.49 ± 5.54	.94 ± .06
16	1	34.17 ± 2.39	.94 ± .03
16	2	60.84 ± 3.42	.93 ± .04
16	4	104.57 ± 5.97	.94 ± .02
16	8	173.73 ± 7.98	.92 ± .04

(d) Shuttle.

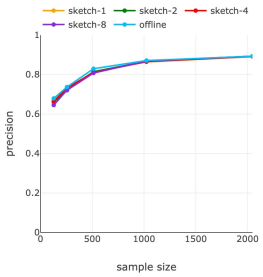
Table 4.6: OneWayCoordinator results for varying number of machines m and sketch size s.



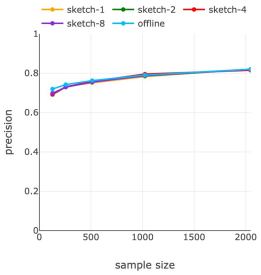
(a) Blobs.



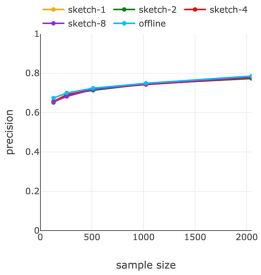
(b) Circles.



(c) Moons.

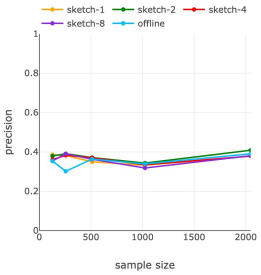


(d) S-Curve.

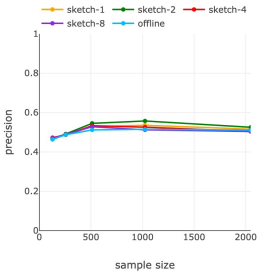


(e) Swiss-Roll

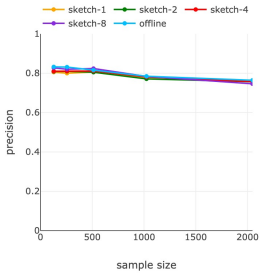
Figure 4.2: TwoWayDistrStreams precision on toy datasets for varying sample and sketch size.



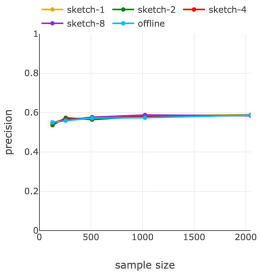
(a) Covtype.



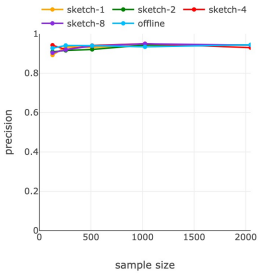
(b) Http.



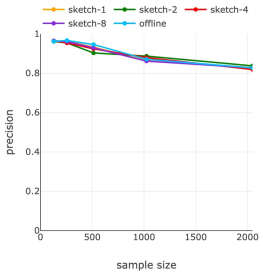
(c) Sat1.



(d) Sat3.

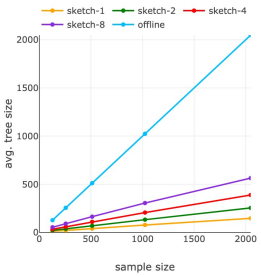


(e) Shuttle.

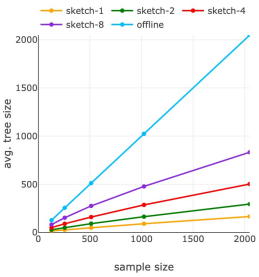


(f) Smtp.

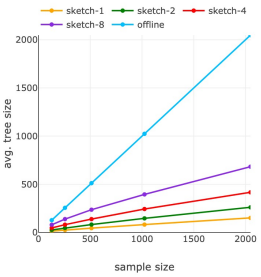
Figure 4.3: TwoWayDistrStreams precision on real datasets for varying sample and sketch size.



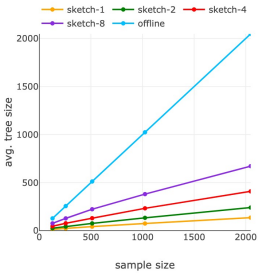
(a) Blobs.



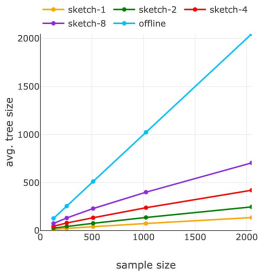
(b) Circles.



(c) Moons.

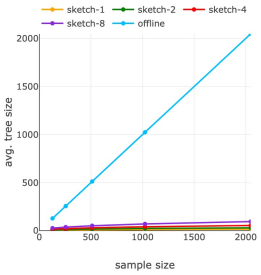


(d) S-Curve.

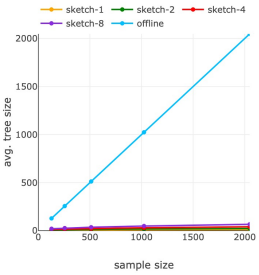


(e) Swiss-Roll.

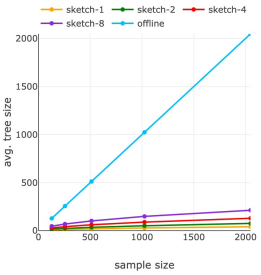
Figure 4.4: TwoWayDistrStreams size on toy datasets for varying sample and sketch size.



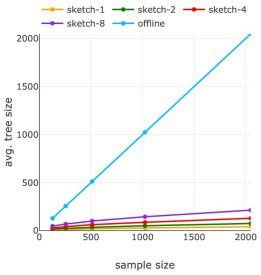
(a) Covtype.



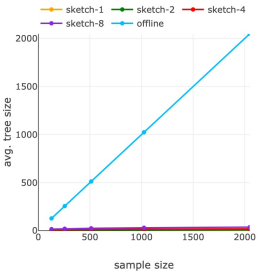
(b) Http.



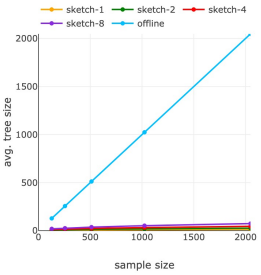
(c) Sat1.



(d) Sat3.

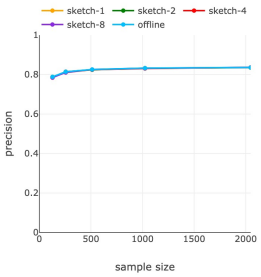


(e) Shuttle.

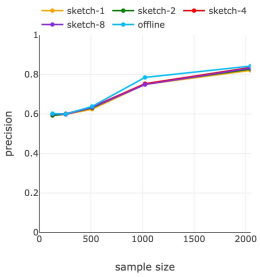


(f) Smtp.

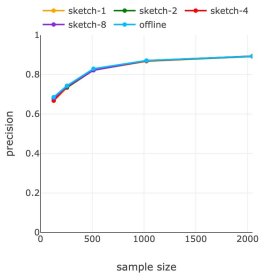
Figure 4.5: TwoWayDistrStreams size on real datasets for varying sample and sketch size.



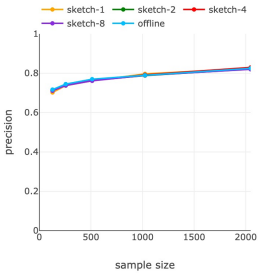
(a) Blobs.



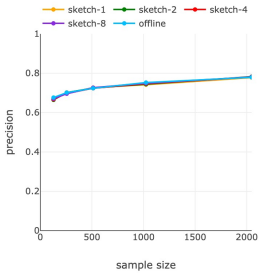
(b) Circles.



(c) Moons.

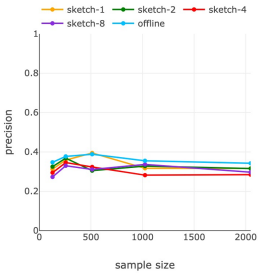


(d) S-Curve.

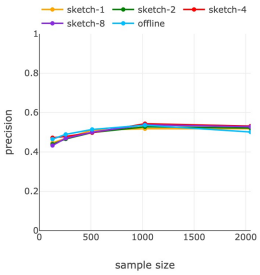


(e) Swiss-Roll.

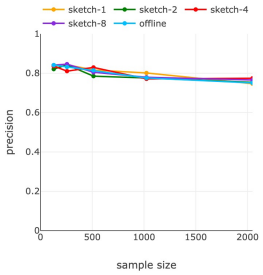
Figure 4.6: OneWayCoordinator precision on toy datasets for varying sample and sketch size.



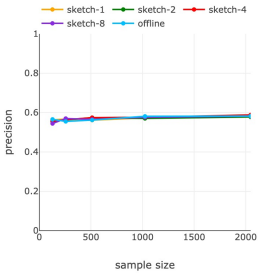
(a) Covtype.



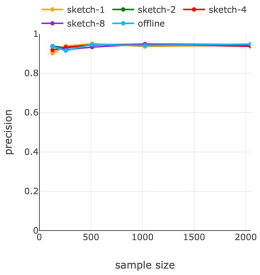
(b) Http.



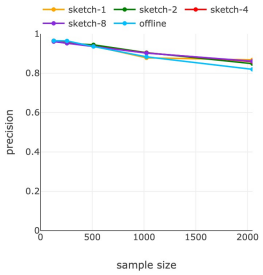
(c) Sat1.



(d) Sat3.

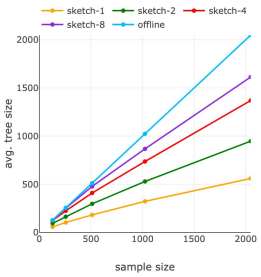


(e) Shuttle.

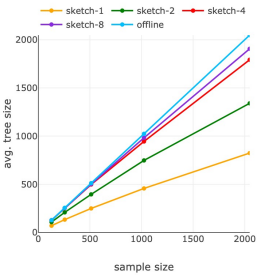


(f) Smtp.

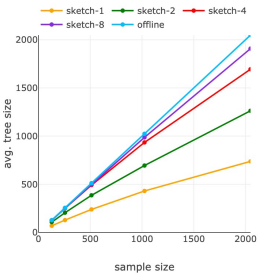
Figure 4.7: OneWayCoordinator precision on real datasets for varying sample and sketch size.



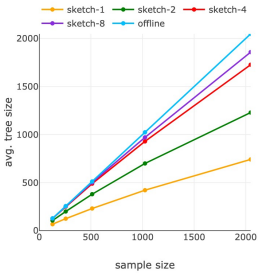
(a) Blobs.



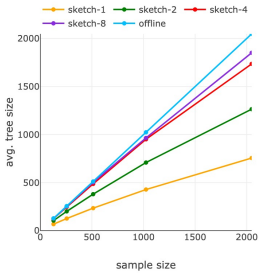
(b) Circles.



(c) Moons.

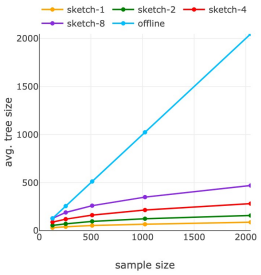


(d) S-Curve.

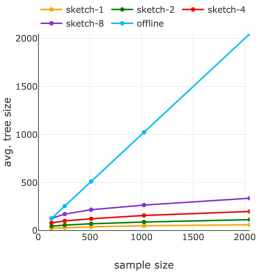


(e) Swiss-Roll.

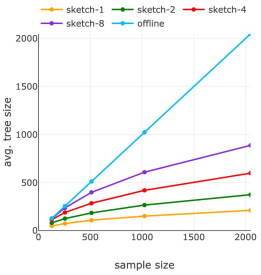
Figure 4.8: OneWayCoordinator size on toy datasets for varying sample and sketch size.



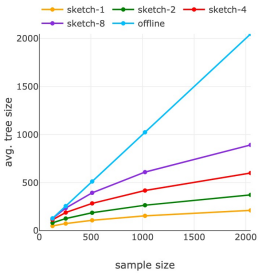
(a) Covtype.



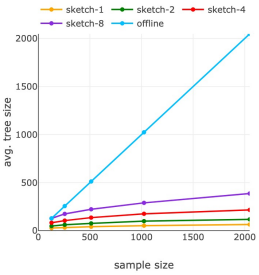
(b) Http.



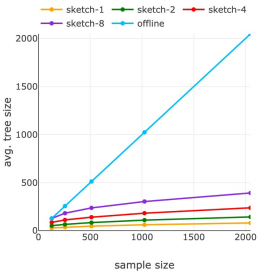
(c) Sat1.



(d) Sat3.



(e) Shuttle.



(f) Smtp.

Figure 4.9: OneWayCoordinator size on real datasets for varying sample and sketch size.

Scalability. To also indicate the further scalability of distributed RSF, figure 4.10 shows the precision; size; and time results for Http when further increasing the number of machines for varying sample sizes. RSF is configured for 32 trees, granularity 4, and sketch size 2. The sample size is varied as 256, 512, 1024, and 2048. The number of machines is varied as 16, 32, 64, and 128.

The results are what you would desire from a distributed algorithm. The running time increases linearly and the memory increases sublinearly with the number of machines. The relative difference between the sample size for the running time and average tree size remains more or less the same. The precision stays the same across all configurations. Note that these experiments were performed on a single 8-core machine. Hence, it is the *overhead* of the distributed algorithm that is measured here, not the performance gains from executing it in parallel (i.e., else there would ideally be a linear *speedup*). Still, the results should serve as an indication of what to expect with the use of an actual cluster.

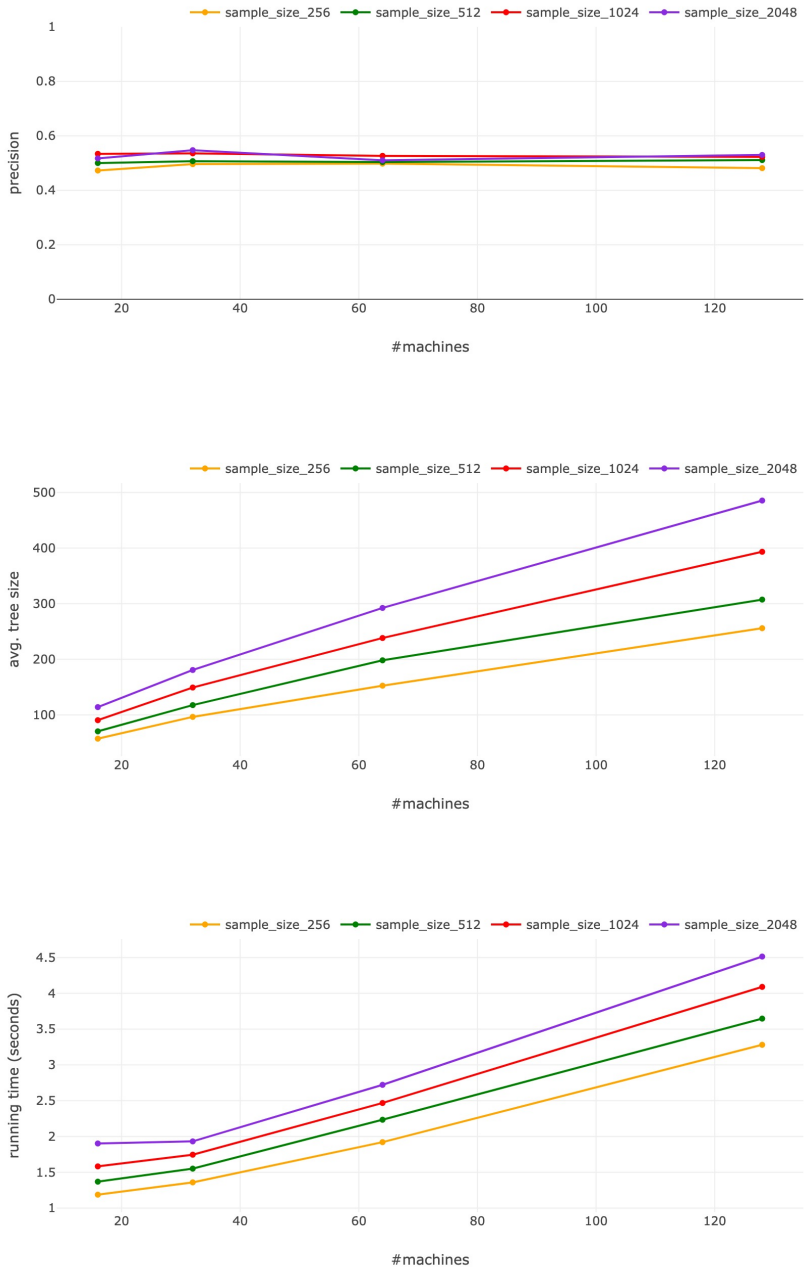


Figure 4.10: OneWayCoordinator scalability on the Http dataset for varying number of machines and sample size.

4.4 Time series

Section 2.3.2 discusses how Autoperiod can be used to automatically detect the period of an input signal, and how shingling can be used to transform the task of anomaly detection to work on patterns of points. This section combines RSF with Autoperiod to perform anomaly detection on shingled time series datasets covering a wide range of use cases.

Numenta Anomaly Benchmark. The Numenta Anomaly Benchmark (NAB) repository [42] is a sizeable repository with time series comprised of the following dataset classes:

- **artificialNoAnomaly:** Synthetic signal data.
- **artificialWithAnomaly:** Synthetic signal data.
- **realAWSCloudwatch:** Server resource metrics.
- **realAdExchange:** Advertisement click rates.
- **realKnownCause:** Real-World scenarios.
- **realTraffic:** Traffic metrics.
- **realTweets:** Company-related tweet frequency.

Each dataset belonging to these classes is one-dimensional with a constant sampling rate, which need not be the case in general. If applicable, multiple signals can be combined into one higher-dimensional signal, enabling anomaly detection on the combined values.

Experiment. Autoperiod is run on the first 1024 points of an incoming stream. The data is shingled with the detected period as the shingle size. Note that the bounding box should be altered accordingly. Then, training and scoring proceed as normal using RSF using (size 2048) window sampling for each tree separately. This includes the points on which Autoperiod is run. RSF is configured with 64 trees of size 512. A dataset is excluded if any of the following applies:

- There are no labels.
- Autoperiod is not able to determine a period.
- The first anomalous event occurs before enough data was seen to start scoring.

Some statistics on the included datasets are shown in table 4.7. The resulting plots are shown in figures 4.11 till 4.15, grouped by each class. The plots are not the result of multiple repetitions.

Discussion. Among the artificialWithAnomaly plots, subfigures 4.11d and 4.11e clearly show the value of using shingling. Viewed as single values, the anomalies do not constitute any abnormalities. The longer valley and increased density of spikes are still at the same height as the other valleys and spikes. However, viewed in the context of the data period, the patterns are completely different; a missing hill in one case and many more spikes in the other. It is the increased absence or presence of these consecutive normal values that can now be detected.

Though the various abnormal signal patterns of artificialWithAnomaly can be detected very well, the realKnownCause plots show practice is a lot messier. The spikes of subfigure 4.12a are regular but rare enough to be repeatedly seen as anomaly. Other data is very flat with many irregular spikes. It should be noted that the provided labels are in no way exhaustive, so many false alarms may actually be applicable, depending on the context. One

class	name	#points	#anomalies	period
realTraffic	occupancy_t4013	2500	2	181
realAWSCloudwatch	ec2_cpu_utilization_24ae8d	4032	2	3
artificialWithAnomaly	art_daily_jumpsup	4032	1	288
realKnownCause	cpu_utilization_asg_misconfiguration	18050	1	12
artificialWithAnomaly	art_load_balancer_spikes	4032	1	35
realTraffic	speed_t4013	2495	2	112
artificialWithAnomaly	art_daily_jumpsdown	4032	1	288
realAWSCloudwatch	ec2_network_in_5abac7	4730	2	264
realAWSCloudwatch	ec2_disk_write_bytes_1ef3de	4730	1	316
realAWSCloudwatch	ec2_cpu_utilization_ac20cd	4032	1	118
realTraffic	speed_6005	2500	1	173
realTweets	Twitter_volume_KO	15851	3	200
realTweets	Twitter_volume_CRM	15902	3	288
realKnownCause	nyc_taxi	10320	5	48
artificialWithAnomaly	art_daily_nojump	4032	1	288
realKnownCause	rogue_agent_key_updown	5315	2	292
artificialWithAnomaly	art_daily_flatmiddle	4032	1	288
realTweets	Twitter_volume_IBM	15893	2	250
realKnownCause	machine_temperature_system_failure	22695	4	218
realTweets	Twitter_volume_FB	15833	2	250
realTweets	Twitter_volume_GOOG	15842	3	244

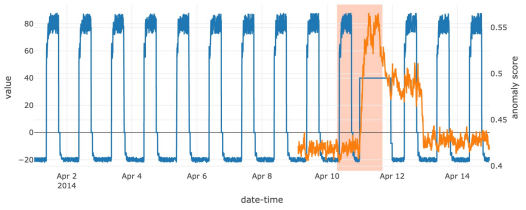
Table 4.7: Statistics of the included NAB datasets.

way to resolve this in practice could be to only sound an alarm when there is an increased anomaly score for some small time window. Another issue is that Autoperiod sometimes still reports a period form some very aperiodic data. These reported periods tend to be quite high.

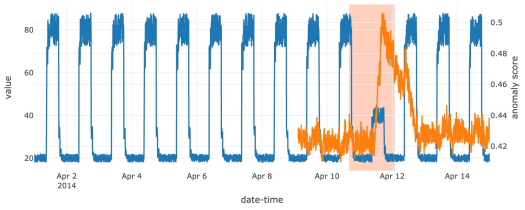
A great example of the combined power of window sampling and shingling is the nyc_taxi plot, measuring taxi volume on a 5min basis. The anomalies correspond to events like New Year's Eve, a marathon, and some severe weather conditions. Each is detected clearly. Interesting is the first peak in the anomaly scores. Although it is not labelled, the distribution visually changes, which is likely due to the transition from summer time to work and school schedules. The weekly anomaly score peaks quickly diminish, showing that besides picking up on the novelty, the time window also quickly catches on to the new pattern. Since the window is not nearly a year long, the same would occur each year, which may be desirable depending on the application.

The results for realTraffic all seem good but are a bit hard to fully judge given the smaller sizes of the datasets. The results on realTweets seem good as well and provide further proof of the effectiveness of using Autoperiod. Tuning the shingle size by hand proved quite difficult, giving much messier results, while there are clear periodicities in the data. These results with the automatically detected period are very clean.

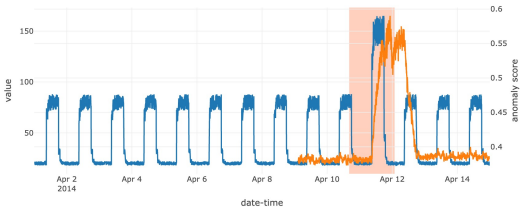
In practice, acquiring the best results comes down to tuning the algorithm's ability to detect and remember short- and long-term patterns. Shingling and window sampling provide indispensable tools to do so. In addition, the automatically detected period using a method like Autoperiod provides a very good starting point for the shingle size. Then, the task comes down to fine-tuning this initial configuration using domain knowledge and introducing additional transformations to handle problem-specific patterns that can occur.



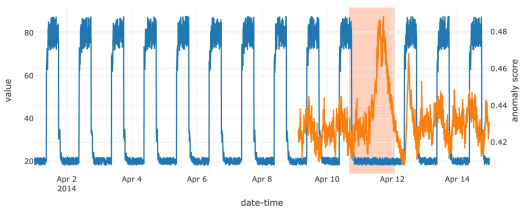
(a) art_daily_flatmiddle.



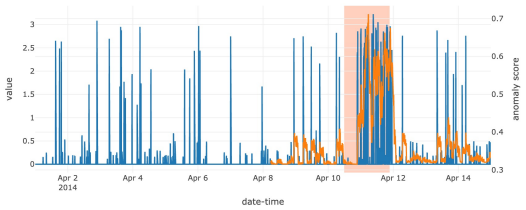
(b) art_daily_jumpsdown.



(c) art_daily_jumpsup.

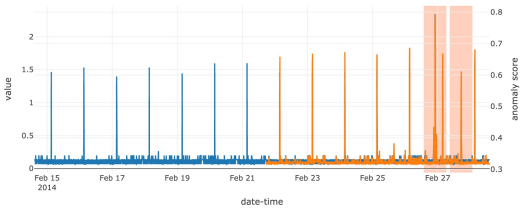


(d) art_daily_nojump.

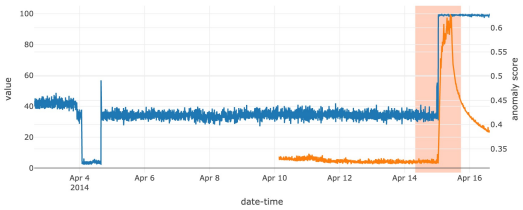


(e) art_load_balancer_spikes.

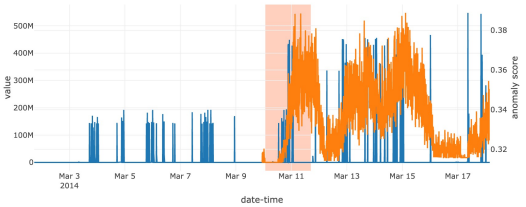
Figure 4.11: artificialWithAnomaly.



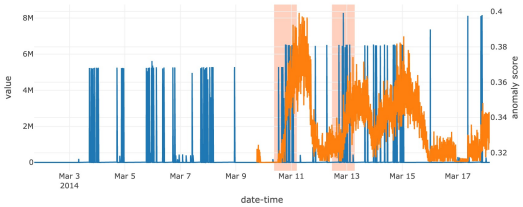
(a) ec2_cpu_utilization_24ae8d.



(b) ec2_cpu_utilization_ac20cd.

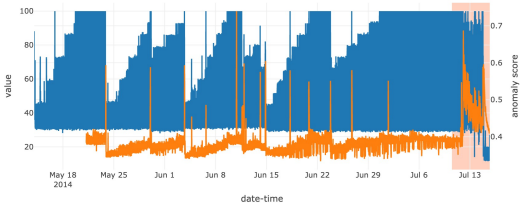


(c) ec2_disk_write_bytes_1ef3de.

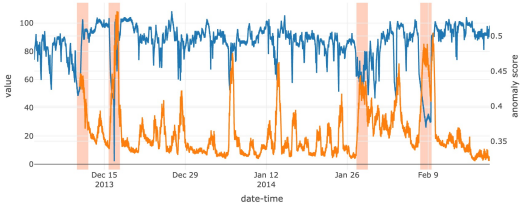


(d) ec2_network_in_5abac7.

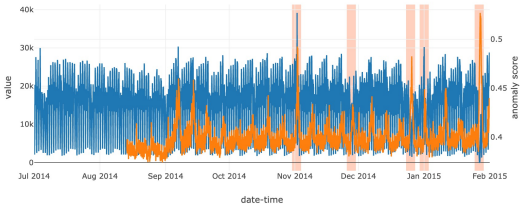
Figure 4.12: realAWSCloudwatch.



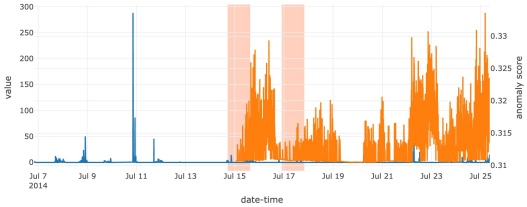
(a) cpu_utilization_asg_misconfiguration.



(b) machine_temperature_system_failure.

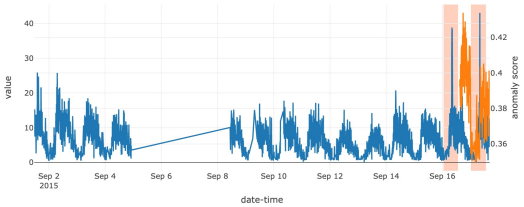


(c) nyc_taxi.

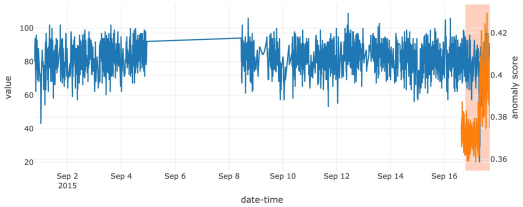


(d) rogue_agent_key_updown.

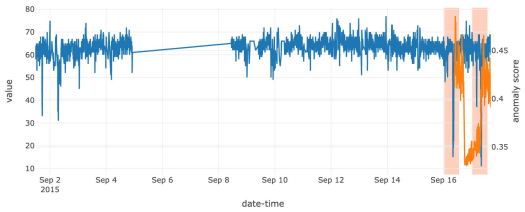
Figure 4.13: realKnownCause.



(a) occupancy_t4013.

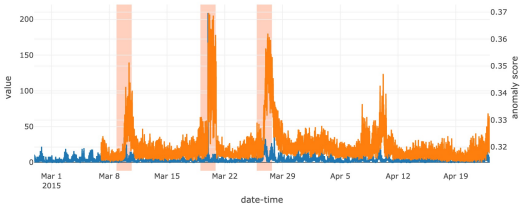


(b) speed_6005.

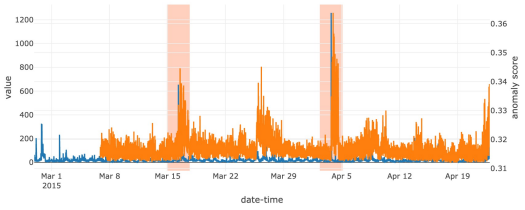


(c) speed_t4013.

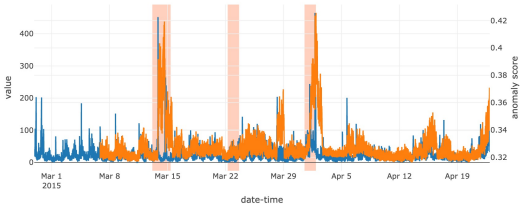
Figure 4.14: realTraffic.



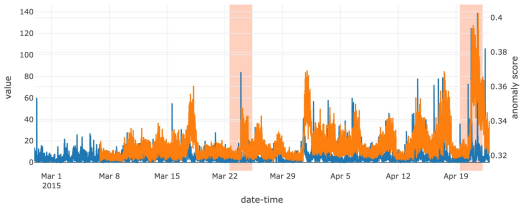
(a) Twitter_volume_CRM.



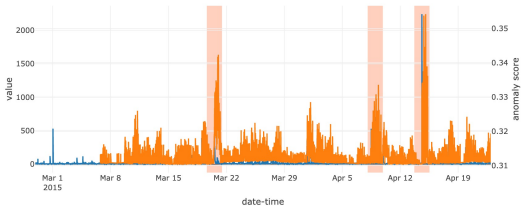
(b) Twitter_volume_FB.



(c) Twitter_volume_GOOG.



(d) Twitter_volume_IBM.



(e) Twitter_volume_KO.

Figure 4.15: realTweets.

4.5 Graphs

Section 2.3.1 discusses how SpotLight embeds a stream of graphs as a stream of points in high-dimensional Euclidean space on which anomaly detection can be performed. This section combines RSF with SpotLight. Another benchmark against RRCF is performed to further compare the performance of both, and delve deeper into the effectiveness of SplotLight. Another experiment applies the approach to Twitter data.

DAPRA. In the paper of its introduction [27], SpotLight is tested on the 1998 DARPA Intrusion Detection Evaluation dataset [42]; available online [1]. This dataset is comprised of the internet traffic of a simulated network in which a wide range of attacks was performed. There are seven weeks of labelled data. After data cleaning and parsing (see [64]), there are 4342431 communication edges. The original experiment aggregates the edges on an hourly basis, marking a graph as anomalous when it contains at least 50 anomalous edges, in this way constructing a stream of graphs from the stream of edges. The graphs are sketched and RRCF is used to perform the final anomaly marking.

Let dt be the aggregation time window and et be the anomalous graph edge threshold. Investigate the performance of split RSF and RRCF (using split sampling) when varying dt (as 300, 1800, and 3600) and et (as 10, 30, and 60). RSF and RRCF are configured with 50 trees of size 256. Spotlight is configured with $K = 50$ and $p = q = 0.2$. These parameters correspond to the original experiment. Figure 4.16 shows how the anomalies are distributed for each combination. To validate the necessity of SpotLight, this experiment also considers two other input streams based on the same subgraphs. The first simply consists of the (1D) edge counts of the input graphs. The second consists of the same SpotLight sketches, but now with L_1-normalization applied to them. This is to investigate which part of the results can be explained by graph size, and what part by the relative weights of the sketch vector entries. All of the results are shown in table 4.8.

The overall shape of the distribution of the anomalies is influenced by the parameters only marginally other than stretching and scaling it. For the same edge thresholds, the number of anomalies is roughly in the same ballpark. This is reflected in the scores, being similar for the same time aggregations, at least for the application of normal SpotLight.

It seems RRCF is better able to detect anomalies for the smallest time aggregation. A smaller time aggregation means more focused events instead of a culmination of small events with a larger time aggregation. The latter may have a dampening effect on the spikes occurring in the sketch vector. In contrast to RSF's deterministic splitting procedure starting from the middle of the bounding box, RRCF's aggressive focus on isolating single points early on may give it the advantage. RSF performs very similarly for all parameters. The two algorithms are quite close for the larger time aggregations, though RRCF takes a slight lead. It may also be problem-specific like with the classification benchmarks. Neither the total edge counts nor the L_1-normalised scores perform as well as normal SpotLight. It is interesting to see that RSF scores slightly better using both of these approaches. Possibly it is the combination where RSF lacks concerning RRCF here.

Finally, as a test of the parameter sensitivity of SpotLight, table 4.9 shows the results for running $dt = 3600, et = 60$ with varying values of K (20, 50, 100, and 200) and $p = q$ (0.01, 0.05, 0.1, and 0.2). Only RSF is used, with the same 50 trees of size 256. The performance turns out to be very close across the whole range of parameter configurations. Only when using $p = q = 0.01$ does the performance drop somewhat for the lower sketch sizes. This shows SpotLight is quite robust and does not need much specific parameter tuning.

dt	et	rocauc	prauc	precision
300	10	0.92 (0.01)	0.79 (0.02)	0.72 (0.01)
300	30	0.94 (0.00)	0.81 (0.03)	0.78 (0.01)
300	60	0.95 (0.00)	0.82 (0.02)	0.79 (0.01)
1800	10	0.82 (0.01)	0.78 (0.01)	0.65 (0.00)
1800	30	0.86 (0.01)	0.79 (0.01)	0.66 (0.02)
1800	60	0.92 (0.01)	0.84 (0.01)	0.73 (0.02)
3600	10	0.79 (0.01)	0.80 (0.01)	0.67 (0.00)
3600	30	0.81 (0.01)	0.79 (0.01)	0.65 (0.01)
3600	60	0.85 (0.01)	0.82 (0.01)	0.68 (0.02)

(a) RRCF-split - normal.

dt	et	rocauc	prauc	precision
300	10	0.79 (0.09)	0.59 (0.12)	0.55 (0.12)
300	30	0.82 (0.09)	0.58 (0.11)	0.54 (0.11)
300	60	0.81 (0.11)	0.58 (0.14)	0.53 (0.14)
1800	10	0.72 (0.06)	0.70 (0.06)	0.62 (0.05)
1800	30	0.77 (0.05)	0.68 (0.06)	0.60 (0.05)
1800	60	0.84 (0.08)	0.73 (0.11)	0.64 (0.10)
3600	10	0.74 (0.02)	0.77 (0.03)	0.66 (0.02)
3600	30	0.75 (0.02)	0.75 (0.03)	0.63 (0.03)
3600	60	0.81 (0.03)	0.76 (0.04)	0.66 (0.04)

(b) RSF-split - normal.

dt	et	rocauc	prauc	precision
300	10	0.73 (0.02)	0.49 (0.01)	0.40 (0.01)
300	30	0.75 (0.02)	0.50 (0.02)	0.44 (0.01)
300	60	0.76 (0.03)	0.50 (0.02)	0.44 (0.01)
1800	10	0.65 (0.01)	0.60 (0.01)	0.49 (0.01)
1800	30	0.66 (0.01)	0.55 (0.01)	0.44 (0.02)
1800	60	0.70 (0.02)	0.56 (0.01)	0.49 (0.01)
3600	10	0.72 (0.01)	0.72 (0.01)	0.63 (0.01)
3600	30	0.75 (0.01)	0.71 (0.01)	0.61 (0.02)
3600	60	0.76 (0.01)	0.68 (0.01)	0.57 (0.02)

(c) RRCF-split - edge counts.

dt	et	rocauc	prauc	precision
300	10	0.78 (0.02)	0.53 (0.01)	0.44 (0.00)
300	30	0.82 (0.02)	0.55 (0.01)	0.46 (0.00)
300	60	0.83 (0.01)	0.56 (0.01)	0.47 (0.00)
1800	10	0.68 (0.02)	0.62 (0.01)	0.57 (0.00)
1800	30	0.72 (0.02)	0.59 (0.01)	0.51 (0.01)
1800	60	0.80 (0.02)	0.62 (0.01)	0.50 (0.00)
3600	10	0.70 (0.01)	0.71 (0.00)	0.64 (0.00)
3600	30	0.72 (0.01)	0.69 (0.00)	0.60 (0.00)
3600	60	0.76 (0.00)	0.68 (0.00)	0.61 (0.01)

(d) RSF-split - edge counts.

dt	et	rocauc	prauc	precision
300	10	0.72 (0.04)	0.32 (0.04)	0.37 (0.08)
300	30	0.70 (0.04)	0.27 (0.03)	0.31 (0.04)
300	60	0.70 (0.03)	0.26 (0.02)	0.30 (0.04)
1800	10	0.74 (0.02)	0.59 (0.03)	0.60 (0.03)
1800	30	0.75 (0.02)	0.52 (0.04)	0.53 (0.02)
1800	60	0.75 (0.03)	0.38 (0.03)	0.44 (0.04)
3600	10	0.68 (0.02)	0.61 (0.02)	0.62 (0.02)
3600	30	0.68 (0.02)	0.55 (0.03)	0.56 (0.02)
3600	60	0.70 (0.02)	0.50 (0.03)	0.51 (0.04)

(e) RRCF-split - L_1-normalisation.

dt	et	rocauc	prauc	precision
300	10	0.73 (0.05)	0.37 (0.06)	0.40 (0.07)
300	30	0.75 (0.05)	0.34 (0.05)	0.36 (0.08)
300	60	0.71 (0.07)	0.30 (0.05)	0.34 (0.07)
1800	10	0.73 (0.03)	0.61 (0.04)	0.60 (0.03)
1800	30	0.76 (0.03)	0.55 (0.06)	0.54 (0.03)
1800	60	0.74 (0.05)	0.42 (0.06)	0.45 (0.06)
3600	10	0.69 (0.02)	0.65 (0.03)	0.63 (0.02)
3600	30	0.68 (0.02)	0.58 (0.03)	0.57 (0.02)
3600	60	0.70 (0.02)	0.54 (0.03)	0.54 (0.02)

(f) RSF-split - L_1-normalisation.

Table 4.8: Results for the DARPA dataset using SpotLight for varying dt and et.

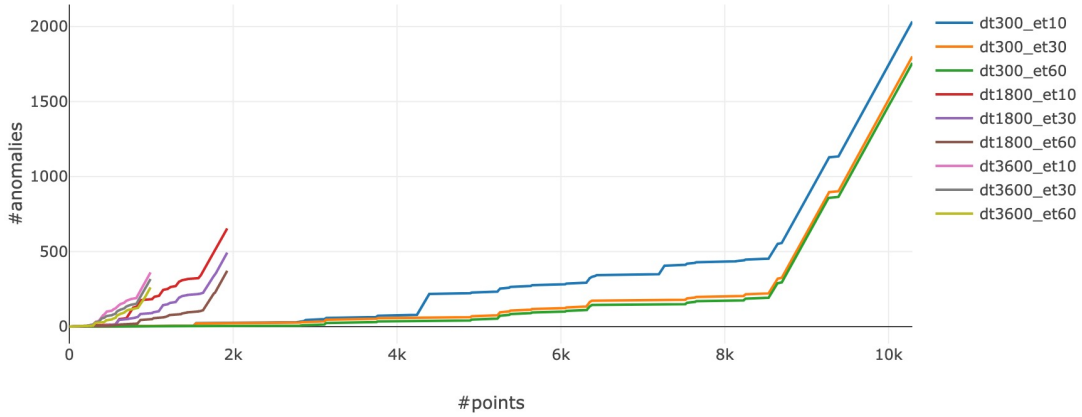


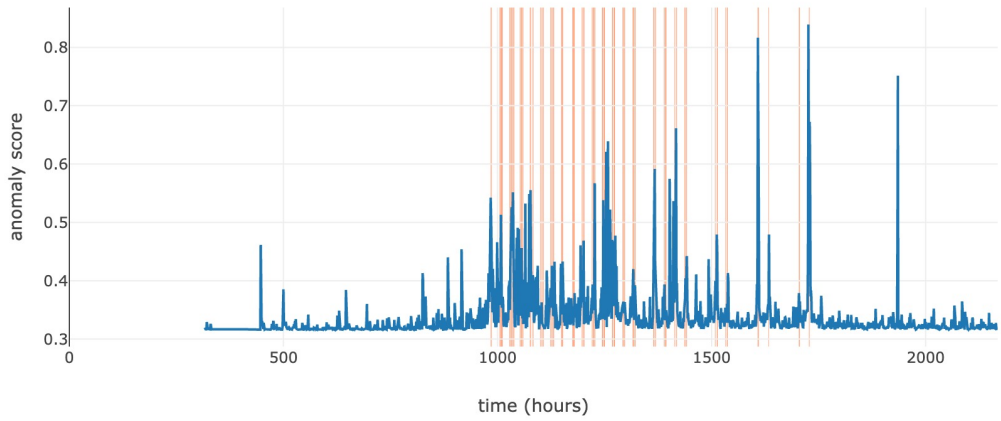
Figure 4.16: Distribution of anomalies for the DARPA dataset for varying dt and et.

K	$p = q$	rocauc	prauc	precision
20	0.01	0.67 (0.10)	0.65 (0.10)	0.48 (0.15)
20	0.05	0.78 (0.08)	0.72 (0.10)	0.63 (0.09)
20	0.1	0.78 (0.06)	0.73 (0.08)	0.63 (0.07)
20	0.2	0.78 (0.05)	0.74 (0.07)	0.64 (0.06)
50	0.01	0.70 (0.09)	0.65 (0.10)	0.55 (0.11)
50	0.05	0.80 (0.02)	0.75 (0.04)	0.65 (0.03)
50	0.1	0.80 (0.04)	0.76 (0.05)	0.65 (0.05)
50	0.2	0.79 (0.04)	0.75 (0.06)	0.65 (0.06)
100	0.01	0.73 (0.08)	0.68 (0.08)	0.59 (0.10)
100	0.05	0.79 (0.06)	0.74 (0.07)	0.64 (0.07)
100	0.1	0.79 (0.04)	0.74 (0.05)	0.64 (0.04)
100	0.2	0.79 (0.05)	0.74 (0.06)	0.64 (0.06)
200	0.01	0.77 (0.06)	0.72 (0.07)	0.63 (0.08)
200	0.05	0.80 (0.04)	0.75 (0.05)	0.65 (0.04)
200	0.1	0.80 (0.04)	0.76 (0.05)	0.65 (0.04)
200	0.2	0.81 (0.02)	0.77 (0.03)	0.66 (0.03)

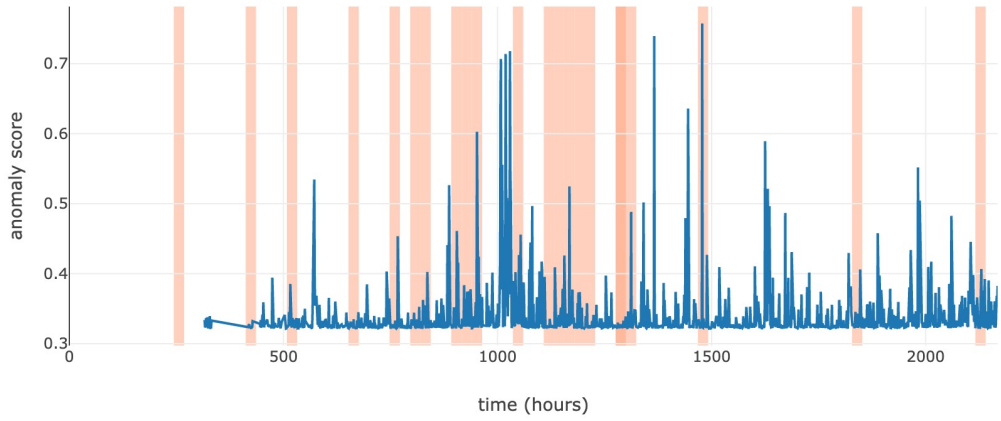
Table 4.9: RSF-split + SpotLight on the DARPA dataset for varying K, p, q.

Twitter. This experiment applies the combination of SpotLight and RSF to the TwitterWorldCup2014 and TwitterSecurity2014 datasets from the Outlier Detection DataSets (ODDS) library [57]. RSF is configured with 64 trees of size 64. It uses reservoir sampling for each tree separately. SpotLight is configured with $K = 50$ and $p = q = 0.2$. Both datasets are labelled, albeit only with dates in the case of TwitterSecurity2014 and with additional time information in the case of TwitterWorldCup2014. The relevant 24h region is shaded for the former, and the relevant 1h region for the latter. The results are shown in figure 4.17.

Other than the time specificity difference, the nature of the labels is also quite different. Whereas TwitterWorldCup2014 is concerned with events like a red card during a soccer match, TwitterSecurity2014 includes events with political weight. This additional specificity difference may explain why the peaks in anomaly scores match up much better in the case of TwitterSecurity2014. In contrast, the peaks for TwitterSecurity2014 are hard to trace back to a specific label. Just using graph edge counts as input (as for the DARPA experiment) here does lead to anomaly score peaks around the same peaks, but also incurs more false alarms, which makes another case for the use of SpotLight.



(a) TwitterWorldCup2014; 1790999 edges; 1791 graphs.



(b) TwitterSecurity2014; 2601834 edges; 1784 graphs.

Figure 4.17: RSF-reservoir + SpotLight on Twitter datasets.

Chapter 5

Conclusions

The goal of this thesis was to extend the use of Random Shift Forest (RSF) to the distributed and streaming settings and explore several ways this can be applied to various types of inputs. This final chapter provides a summary of the results and lists the limitations of this study with suggestions for future work.

Summary. Chapter 3 formalises RSF as a version of Isolation Forest (iForest) [45] where instead of splitting the subsample at random values, the input is shifted randomly and the splits are performed deterministically. In addition, rather than a recursive definition, a more general incremental definition is given with insert and delete operators, while retaining the same complexity bounds in terms of both storage and running time. RSF is also shown to reduce the bias that iForest suffers from its axis-aligned splits. The operations allow for constructing and updating RSF in a streaming fashion using either reservoir or window sampling. The deterministic elements allow for the general application of RSF in the distributed domain, for which various versions of RSF were proposed and analysed. In addition, it allows for the use of sketching the underlying data structure of RSF. A simple sketch is proposed to reduce the communication complexity of distributed RSF.

Chapter 4 provides experiments to empirically verify that RSF is on par with other state-of-the-art unsupervised isolation-based anomaly detection methods for various toy and real datasets. The implementation of two distributed versions of RSF are verified to have the same detection capability as offline RSF. In addition, the sketching ability is put to use with the simple sketch method proposed. The sketch is shown to drastically reduce the average RSF tree size without degrading performance. On a wide range of different time series datasets, the usefulness of shingling in tandem with window sampling is argued as a way to detect short-and long term signal pattern breakage. Autoperiod [66] is proven to successfully automate the process of picking an appropriate shingle size. Finally, SpotLight [27] was shown to enable the detection of anomalous subgraphs, albeit with varying success. Yet, near real-time detection using small edge aggregation windows and detection for social media data were presented as possible use cases.

Limitations and suggestions. For Extended Isolation Forest (EIF), rotation and hyperplane splitting were proposed. Although hyperplanes seemed to work best, applying rotation is compatible with the setup of RSF, and may introduce additional favourable randomness. Every tree then applies a random shift and a random rotation matrix to its input. Compared to sampling the direction vector for the hyperplanes, obtaining such a rotation matrix is trickier. The work by Blaser and Fryzlewicz [10] provides a possible method. Early efforts of implementing such a rotational RSF are included with the other code [35], but were not fully worked out or investigated.

It was not sufficiently shown where reservoir sampling is of the essence. It was argued it is important in absence of clean training data, but this was not explicitly verified. Other than that, only the insertion-only and sliding-window streaming models were considered. Dynamic streaming was not considered. Maintaining a stream sample when both insertions and deletions occur is a lot trickier. Techniques to sample dynamic streams do exist; see for instance [29] (strict turnstile model) and [8] (non-strict turnstile model). The advantage of the RSF algorithm presented here is that all methods can be used as long as the sample can be synced with RSF through the atomic inserts and delete operations.

Autoperiod was only applied to the beginning of the time series datasets. The periodicity structure and thus the shingle size may very a result of a change in the data distribution. This would also mean a varying input dimension. What's more, input bounds could also be subject to change. It was not discussed how RSF could adapt to these situations. One answer may be found in the work of Tan; Ting; and Liu [61], where a jumping window approach is used. Here, each window of points is scored but also used as the input for the construction of a new forest for the next window, replacing the current. This gives the freedom to reconfigure the algorithm completely, but it does introduce a granularity trade-off. In terms of runtime, scoring remains the same, but reconstruction happens for each window, though still incrementally in logarithmic time per element.

It was argued RSF's determinism is essential for being able to distribute it, but cases where other methods actually fail were not presented. The version of RSF for the distributed streaming model used a 2-pass algorithm, while a 1-pass algorithm would be much preferred in always-online settings. This could be done by continuing the algorithm used for the original sampling, but this has not been worked out further. Running RSF on an actual compute cluster was not performed either. For such work, see for instance [32], where EIF is deployed on a Kubernetes cluster (popular runtime for distributed container-based systems).

RSF was shown to be at the centre of a versatile anomaly detection framework. More work remains to verify its usefulness for other types of data, improve its robustness, and make it work in more dynamic contexts.

Bibliography

[1] 1998 darpa intrusion detection evaluation. https://www.ll.mit.edu/r-d/datasets/1998-darpa-intrusion-detection-evaluation-dataset. 53

[2] Covertype data set. https://archive.ics.uci.edu/ml/datasets/covertype. 35

[3] Kdd cup 1999 data. http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html. 35

[4] Statlog (landsat satellite) data set. https://archive.ics.uci.edu/ml/datasets/Statlog+(Landsat+Satellite). 36

[5] Statlog (shuttle) data set. https://archive.ics.uci.edu/ml/datasets/Statlog+(Shuttle). 36

[6] Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 20–29, 1996. 3

[7] Amazon Web Services. Random cut forest by AWS. https://github.com/aws/random-cut-forest-by-aws, 2021. 34

[8] Neta Barkay, Ely Porat, and Bar Shalem. Efficient sampling of non-strict turnstile data streams. *Theoretical Computer Science*, 590:106–117, 2015. 59

[9] Liron Bergman and Yedid Hoshen. Classification-based anomaly detection for general data. *arXiv preprint arXiv:2005.02359*, 2020. 2

[10] Rico Blaser and Piotr Fryzlewicz. Random rotation ensembles. *The Journal of Machine Learning Research*, 17(1):126–151, 2016. 12, 58

[11] Richard J Bolton, David J Hand, et al. Unsupervised profiling methods for fraud detection. *Credit scoring and credit control VII*, pages 235–255, 2001. 7

[12] Markus M. Breunig, Hans-Peter Kriegel, Raymond T. Ng, and Jörg Sander. Lof: Identifying density-based local outliers. SIGMOD '00, page 93–104, New York, NY, USA, 2000. Association for Computing Machinery. 8, 9

[13] Mete Çelik, Filiz Dadaşer-Çelik, and Ahmet Şakir Dokuz. Anomaly detection in temperature data using dbscan algorithm. In *2011 international symposium on innovations in intelligent systems and applications*, pages 91–95. IEEE, 2011. 9

[14] Raghavendra Chalapathy, Aditya Krishna Menon, and Sanjay Chawla. Anomaly detection using one-class neural networks. *arXiv preprint arXiv:1802.06360*, 2018. 8

[15] V. Chandola, A. Banerjee, and Kumar. Anomaly detection: A survey. In *Computing Surveys 41, 3*, pages 1–58, 2009. 9

[16] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM computing surveys (CSUR)*, 41(3):1–58, 2009. iv, 7, 8

[17] Yung-Yu Chung, Srikanta Tirthapura, and David P Woodruff. A simple message-optimal algorithm for random sampling from a distributed stream. *IEEE Transactions on Knowledge and Data Engineering*, 28(6):1356–1368, 2016. 3, 15, 18

[18] Graham Cormode, Shanmugavelayutham Muthukrishnan, Ke Yi, and Qin Zhang. Continuous sampling from distributed streams. *Journal of the ACM (JACM)*, 59(2):1–25, 2012. iv, 4, 5

[19] Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Maintaining stream statistics over sliding windows. *SIAM journal on computing*, 31(6):1794–1813, 2002. 3

[20] Qi Ding and Eric D Kolaczyk. A compressed pca subspace method for anomaly detection in high-dimensional data. *IEEE Transactions on Information Theory*, 59(11):7419–7433, 2013. 9

[21] Dheeru Dua and Casey Graff. UCI machine learning repository, 2017. 35

[22] Nick Duffield, Patrick Haffner, Balachander Krishnamurthy, and Haakon Ringberg. Rule-based anomaly detection on ip flows. In *IEEE INFOCOM 2009*, pages 424–432. IEEE, 2009. 8

[23] Andrew F. Emmott, Shubhomoy Das, Thomas Dietterich, Alan Fern, and Weng-Keen Wong. Systematic construction of anomaly detection benchmarks from real data. In *ACM SIGKDD Workshop on Outlier Detection and Description*, pages 16–21, 2013. 9, 10

[24] Funda Ergun, Hossein Jowhari, and Mert Sağlam. Periodicity in streams. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, pages 545–559. Springer, 2010. 17

[25] Hugo Jair Escalante. A comparison of outlier detection algorithms for machine learning. *Programming and Computer Software*, 01 2005. 7

[26] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd*, volume 96, pages 226–231, 1996. 8, 9

[27] Dhivya Eswaran, Christos Faloutsos, Sudipto Guha, and Nina Mishra. Spotlight: Detecting anomalies in streaming graphs. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1378–1386, 2018. iv, 1, 3, 15, 16, 53, 58

[28] Romain Fontugne, Pierre Borgnat, Patrice Abry, and Kensuke Fukuda. Mawilab: combining diverse anomaly detectors for automated anomaly labeling and performance benchmarking. In *Proceedings of the 6th International COnference*, pages 1–12, 2010. 9

[29] Gereon Frahling, Piotr Indyk, and Christian Sohler. Sampling in dynamic data streams and applications. In *Proceedings of the twenty-first annual symposium on Computational geometry*, pages 142–149, 2005. 3, 59

[30] Markus Goldstein and Andreas Dengel. Histogram-based outlier score (hbos): A fast unsupervised anomaly detection algorithm. *KI-2012: poster and demo track*, 9, 2012. 9

[31] Sudipto Guha, Nina Mishra, Gourav Roy, and Okke Schrijvers. Robust random cut forest based anomaly detection on streams. In *International conference on machine learning*, pages 2712–2721. PMLR, 2016. iv, 1, 9, 13, 14, 17, 28

[32] Sahand Hariri and Matias Carrasco Kind. Batch and online anomaly detection for scientific applications in a kubernetes environment. In *Proceedings of the 9th Workshop on Scientific Cloud Computing*, pages 1–7, 2018. 59

[33] Sahand Hariri, Matias Carrasco Kind, and Robert J Brunner. Extended isolation forest. *IEEE Transactions on Knowledge and Data Engineering*, 33(4):1479–1489, 2019. iv, 1, 9, 11, 12, 13, 14

[34] R. Heek. Extended isolation forest. `https://github.com/rubenheek/extended-isolation-forest`, 2022. 34

[35] R. Heek. Random shift forest. `https://github.com/rubenheek/rsf`, 2022. 34, 58

[36] Piotr Indyk. Algorithms for dynamic geometric problems over data streams. In *Proceedings of the thirty-sixth annual ACM Symposium on Theory of Computing*, pages 373–380, 2004. 3

[37] Wen Jin, Anthony KH Tung, Jiawei Han, and Wei Wang. Ranking outliers using symmetric neighborhood relationship. In *Pacific-Asia conference on knowledge discovery and data mining*, pages 577–593. Springer, 2006. 9

[38] Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for mapreduce. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*, pages 938–948. SIAM, 2010. 3

[39] Hans-Peter Kriegel, Peer Kröger, Erich Schubert, and Arthur Zimek. Loop: local outlier probabilities. In *Proceedings of the 18th ACM conference on Information and knowledge management*, pages 1649–1652, 2009. 9

[40] Heesung Kwon and Nasser M Nasrabadi. Kernel rx-algorithm: A nonlinear anomaly detector for hyperspectral imagery. *IEEE transactions on Geoscience and Remote Sensing*, 43(2):388–397, 2005. 9

[41] Nikolay Laptev, Saeed Amizadeh, and Ian Flint. Generic and scalable framework for automated time-series anomaly detection. In *Proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1939–1947, 2015. 2

[42] Alexander Lavin and Subutai Ahmad. Evaluating real-time anomaly detection algorithms–the numenta anomaly benchmark. In *2015 IEEE 14th international conference on machine learning and applications (ICMLA)*, pages 38–44. IEEE, 2015. 35, 47, 53

[43] Rikard Laxhammar. Anomaly detection for sea surveillance. In *2008 11th international conference on information fusion*, pages 1–8. IEEE, 2008. 9

[44] Kun-Lun Li, Hou-Kuan Huang, Sheng-Feng Tian, and Wei Xu. Improving one-class svm for anomaly detection. In *Proceedings of the 2003 international conference on machine learning and cybernetics (IEEE Cat. No. 03EX693)*, volume 5, pages 3077–3081. IEEE, 2003. 8

[45] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. Isolation forest. In *2008 eighth ieee international conference on data mining*, pages 413–422. IEEE, 2008. iv, 1, 9, 10, 11, 58

[46] N. Mandery. Extended isolation forest. `https://github.com/nmandery/extended-isolation-forest`, 2022. 34

[47] Steven Mascaro, Ann E Nicholso, and Kevin B Korb. Anomaly detection in vessel tracks using bayesian networks. *International Journal of Approximate Reasoning*, 55(1):84–98, 2014. 8

[48] Gerhard Münz, Sa Li, and Georg Carle. Traffic anomaly detection using k-means clustering. In *GI/ITG Workshop MMBnet*, volume 7, page 9, 2007. 9

[49] Shanmugavelayutham Muthukrishnan et al. Data streams: Algorithms and applications. *Foundations and Trends® in Theoretical Computer Science*, 1(2):117–236, 2005. 3

[50] Spiros Papadimitriou, Hiroyuki Kitagawa, Phillip B Gibbons, and Christos Faloutsos. Loci: Fast outlier detection using the local correlation integral. In *Proceedings 19th international conference on data engineering (Cat. No. 03CH37405)*, pages 315–326. IEEE, 2003. 9

[51] Athanasios Papoulis. *Signal analysis*. Mcgraw-Hill College, 1977. 18

[52] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011. 24, 34

[53] Leonid Portnoy, Eleazar Eskin, and Salvatore Stolfo. Intrusion detection with unlabeled data using clustering. 11 2001. 7

[54] Bruno R.. Preiss. *Data Structure and Algorithms: With Object-oriented Design Patterns in Java*. John Wiley & Sons, 1999. 11

[55] Tom Puech, Matthieu Boussard, Anthony D'Amato, and Gaëtan Millerand. A fully automated periodicity detection in time series. In *International Workshop on Advanced Analysis and Learning on Temporal Data*, pages 43–54. Springer, 2019. 18

[56] Marie-Julie Rakotosaona, Vittorio La Barbera, Paul Guerrero, Niloy J Mitra, and Maks Ovsjanikov. Pointcleannet: Learning to denoise and remove outliers from dense point clouds. In *Computer Graphics Forum*, volume 39, pages 185–203. Wiley Online Library, 2020. 7

[57] Shebuti Rayana. ODDS library, 2016. 56

[58] Osman Salem, Alexey Guerassimov, Ahmed Mehaoua, Anthony Marcus, and Borko Furht. Anomaly detection in medical wireless sensor networks using svm and linear regression models. *International Journal of E-Health and Medical Communications (IJEHMC)*, 5(1):20–45, 2014. 9

[59] Durgesh Samariya and Amit Thakkar. A comprehensive survey of anomaly detection algorithms. *Annals of Data Science*, pages 1–22, 2021. 9

[60] P. N. Tan, M. Steinbach, and V. Kumar. Introduction to data mining. *Addison-Wesley*, 2005. 9

[61] Swee Chuan Tan, Kai Ming Ting, and Tony Fei Liu. Fast anomaly detection for streaming data. In *Twenty-second international joint conference on artificial intelligence*, 2011. 59

[62] Jian Tang, Zhixiang Chen, Ada Wai-Chee Fu, and David W Cheung. Enhancing effectiveness of outlier detections for low density patterns. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 535–548. Springer, 2002. 9

[63] Alaa Tharwat. Classification assessment methods. *Applied Computing and Informatics*, 2020. iv, 5, 6, 7

[64] ToshikiShawn. spotlight_anomaly_detection. `https://github.com/ToshikiShawn/spotlight_anomaly_detection`, 2019. 53

[65] Thijs Visser. Isolation-based anomaly detection algorithms for distributed data streams. 2021. 1, 24, 27, 31

[66] Michail Vlachos, Philip Yu, and Vittorio Castelli. On periodicity detection and structural periodic similarity. In *Proceedings of the 2005 SIAM international conference on data mining*, pages 449–460. SIAM, 2005. iv, 1, 3, 15, 17, 18, 58