dSalmon: High-Speed Anomaly Detection for Evolving Multivariate Data Streams

Alexander Hartl(✉), Félix Iglesias, and Tanja Zseby

TU Wien—Institute of Telecommunications, 1040 Wien, Austria
{alexander.hartl,felix.iglesias,tanja.zseby}@tuwien.ac.at

Abstract. We introduce dSalmon, a highly efficient framework for outlier detection on streaming data. dSalmon can be used with both Python and C++, meeting the requirements of modern data science research. It provides an intuitive interface and has almost no package dependencies. dSalmon implements main stream outlier detection approaches from literature. By using pure C++ in its core and making the most of available parallelism, data is analyzed with superior processing speed.
We describe design decisions and outline the software architecture of dSalmon. Additionally, we perform thorough evaluations on benchmarking datasets to measure execution time, memory requirements and energy consumption when performing outlier detection. Experiments show that dSalmon requires substantially less resources and in most cases is able to process datasets between one and three orders of magnitude faster than established Python implementations.

Keywords: Outlier detection · Data streams · Unsupervised learning · Python · C++

1 Introduction

In a world of ever-increasing transmission rates, performing knowledge discovery on data streams becomes more and more challenging. A particularly important and well-known data mining task is detecting anomalies and outliers. A variety of methods have been proposed for Outlier Detection (OD) on static datasets [8,17], but also for online OD on a live stream of arriving data samples [18].

While algorithm implementations with low processing speed might pose a problem for practical online processing of data streams, it is even more challenging for researchers. Performing feature selection or fitting algorithm parameters using, e.g., grid search, randomized search or Bayesian optimization, involves

This work was supported by the project MALware cOmmunication in cRitical Infrastructures (MALORI), funded by the Austrian security research program KIRAS of the Federal Ministry for Agriculture, Regions and Tourism (BMLRT) under grant no. 873511.

E. Kalyvianaki and M. Paolieri (Eds.): VALUETOOLS 2023, LNICST 539, pp. 153–169, 2024.
https://doi.org/10.1007/978-3-031-48885-6_10

a vast amount of algorithm runs on a captured data stream, which naturally should cover a period that is as long as possible. Additionally, researchers often have the requirement to test algorithms on multiple datasets. Hence, the computational ability to process a data stream in real time is far from sufficient for research and slow algorithm implementations become a burden for researchers or even prevent them from considering specific methods for a given task.

With this paper, we specifically target *evolving* data streams, i.e. streams whose properties and structures in feature space change over time. Models therefore have to be continuously updated or retrained to avoid declining detection performance due to outdated models. This characteristic of data streams is also frequently referred to as *concept drift*. When it comes to algorithm run times, we note a basic systematic difference for processing evolving streaming data compared to the processing of static datasets considering opportunities for batch processing in scripting languages. Due to the lack of an inherent ordering, for static datasets, blocks of data samples can be evaluated against one and the same model, in many cases yielding opportunities for batch processing and, hence, fast processing. The same cannot be said about evolving streaming data. Since algorithms continuously adapt to newly seen data, the model relevant for mining data sample n is potentially influenced by all data samples $0, \ldots, n-1$. Any attempt for faster processing by evaluating a block of data against the same model would therefore yield inaccurate results compared to production use, where data samples are processed one at a time at the time of their arrival.

We present our **D**ata **S**tream **A**nalysis Algorith**m**s f**o**r the Impatie**n**t (dSalmon)[1], a framework for performing OD on multivariate evolving streams of data, which has been specifically designed to process data as efficiently as possible with respect to both execution time and memory footprint. dSalmon provides a simple and intuitive Python interface to allow rapid development by data scientists, but performs processing in C++, achieving substantial performance benefits compared to existing implementations. It is easily extendable by deploying software for automatically generating boilerplate code and has almost no package dependencies. We perform a thorough comparison of dSalmon to PySAD [35], the only Python framework for performing OD on streaming data to date. To provide a comprehensive evaluation, we measure run time, memory usage and energy consumption when applying stream OD algorithms on three different publicly available benchmarking datasets. Our findings show that dSalmon provides substantial benefits for all resources. Execution time improvements of up to three orders of magnitude can be obtained with dSalmon.

The remainder of this paper is structured as follows. After highlighting related software projects in Sect. 2, we provide a brief introduction into the foundations of modern stream OD approaches in Sect. 3, particularly considering challenges when processing high-rate data streams. In Sect. 4, we then discuss design objectives, our resulting architectural design and the interface of dSalmon. To demonstrate that substantial performance benefits can be obtained with our

[1] https://github.com/CN-TU/dSalmon.

deployed architecture, we proceed in Sect. 5 with a comprehensive experimental evaluation and comparison of resource consumption when using our framework. Section 6 concludes the paper.

2 Related Work

The most important software project related to dSalmon is the recent PySAD [35] framework, which similarly targets OD on streaming data in Python. PySAD provides several methods for OD on data streams and is entirely written in Python. Some of the outlier detectors PySAD provides wrap existing Python solutions, like, e.g., from the PyOD [36] framework or scikit-learn [24]. In this paper, we compare runtime performance with PySAD, since algorithms implemented in PySAD are similar to the ones we provide.

For OD tasks, PyOD [36] or scikit-learn [24] can also be used directly. PyOD is a popular Python package that provides several methods for OD. Unlike dSalmon, it targets methods for processing static datasets rather than streaming data. Several outlier detectors for static datasets are also provided by scikit-learn, the most popular Python framework for machine learning and data mining.

A well-known software project for processing streaming data is the MOA [7] framework implemented in Java. Algorithms provided by MOA are not limited to OD, but cover several fields of data mining like clustering and classification. Outlier detectors for streaming data implemented in MOA are, however, limited to a distance-based outlier definition and provide only binary labels instead of outlier scores. In dSalmon, we additionally implement several recent approaches for stream OD like ensemble-based methods. MOA is meant to be used as a stand-alone application rather than a programming library. Since implemented algorithms additionally differ severely from algorithms implemented in dSalmon, we do not include MOA in our experimental evaluations.

In the Java community, also the ELKI [27] framework is worth mentioning. ELKI provides a comprehensive selection of data mining algorithms. However, similar to PyOD and scikit-learn, it focuses on the processing of static datasets instead of streaming data processing.

3 Outlier Detection on Evolving Data Streams

Detection of outliers in evolving data streams is an important data mining problem. Several techniques have been proposed in the literature [18]. Coarsely classified, OD can be performed based on distances to nearest neighboring samples [3,16,19,34], deploying histograms [25], relying on tree structures [12,29] or by performing density estimations in feature space [22,26].

In data stream analysis, the underlying technique for handling concept drift is as critical as computing outlier scores. This aspect of algorithm construction is of substantial importance, since it determines whether the memory length, i.e. the duration for which patterns in the data stream should be remembered, has a strong influence on the algorithm's resource consumption. Particularly in the context of high-rate processing, this property if of substantial relevance.

	SW-DBOR	SW-KNN	SW-LOF	LODA [25]	RS-Hash [26]	RRCF [12]	HS-Trees [29]	xStream [22]	SDOstream [16]
Windowing mechanism	SW	SW	SW	SW	SW	SW	RW	RW	EW
Constant space complexity	×	×	×	×	×	×	✓	✓	✓
Constant time complexity	×	×	×	×	✓	×	✓	✓	✓
Ensemble-based	×	×	×	✓	✓	✓	✓	✓	×
Parallelizable in dSalmon	×	×	×	✓	✓	✓	✓	✓	×

Fig. 1. Methods for streaming OD from related work available in dSalmon.

For handling concept drift, several methods follow a *sliding window* (SW) approach [3,12,19,25,26,34], where the most recently seen N data samples are stored and used for assessing outlierness of newly arriving data samples, $N \in \mathbb{N}$ denoting the window length. The use of a SW has the effect that space complexity depends linearly on memory length, rendering SW-based algorithms impractical for high-rate data streams. In addition, as a general rule, time complexity is significantly influenced by memory length.

For this reason, some modern OD methods adopt a *reference window* (RW) approach [22,29]. In this case, a model is trained from observed data samples without performing outlier scoring on the observed data. After having observed N data samples, the trained model is deemed the RW and used for scoring outlierness of newly arriving data samples, while a new model is trained from newly observed data. Hence, for scoring outlierness of points $kN \dots (k+1)N$ with $k \in \mathbb{N}$ and the window length $N \in \mathbb{N}$, data samples $(k-1)N \dots kN-1$ are used as RW. RW methods typically achieve space and time complexity $\mathcal{O}(1)$ with respect to memory length.

Another approach for achieving $\mathcal{O}(1)$ complexity is the use of an *exponential window* (EW) [16,26]. In this case, models use an exponentially weighted moving average for internal counters, avoiding the need to store data samples.

In Fig. 1, we show OD methods implemented in dSalmon and several characteristics relevant for execution time.

4 Architectural Design and Interface

We now describe how we engineered dSalmon to optimize usability for research on streaming data processing. To motivate our architectural design, we start by depicting software goals and resulting design decisions.

4.1 Design Decisions

We primarily target researchers working with streaming data who aim to develop or optimize systems and algorithms and therefore possess offline datasets of captured data streams. Here we present the objectives that motivated our design decisions for dSalmon.

- **High Speed Data Processing.** A primary goal is the optimization of run times of the algorithms provided by dSalmon. This is especially important for researchers who conduct tests with different algorithms and parameters, aiming to make an informed decision about the best parameterization.
  Design Decision: To achieve high-speed processing, the core of dSalmon is implemented in C++. Furthermore, a spatial indexing data structure is used to reduce execution times of nearest neighbors and range queries.
- **Straightforward Usability.** In the field of data science, Python is a commonly used language. Many tools provide Python interfaces and researchers often develop algorithms in Python. While largely implemented in C++, we thus aimed to provide an interface that is familiar in the Python community.
  Design Decision: For outlier detectors, we adhere to the known interface of scikit-learn, which also has been adopted by PyOD.
- **Processing of Recorded Data.** While in the application phase data samples have to be processed one at a time, during algorithm development and parameter tuning researchers commonly can make use of datasets consisting of previously collected streaming data. If desired, dSalmon allows providing blocks of data as input to achieve superior processing speed. In such cases, to accomplish a behavior equivalent to application phases, the implementation must guarantee to be invariant of the used block size.
  Design Decision: To provide block size invariance, we process block samples sequentially within our fast C++ backend. Hence, block size has no effect on the returned results.
- **Support for Efficient Ensemble Learning.** In recent research, it has become common to construct data mining algorithms by pooling the results of an ensemble of weak learners, providing opportunities for embarrassingly parallel processing. These opportunities for parallelization have to be passed on the user, allowing a substantial speedup on modern computing hardware.
  Design Decision: We leverage available parallelism in the core of dSalmon, allowing the user to simply set an `n_jobs` parameter to reduce run time.
- **Reproducibility.** To support reproducibility of results, results obtained from randomized algorithms have to be parameterized by a random seed, so that results are deterministic and reproducible when providing the same seed value. Changing the used block size or the number of parallel computing threads has to leave the obtained results unaffected.
  Design Decision: We support parameterization of randomized algorithms by a random seed and engineered algorithm implementations to be invariant of block size or requested parallelism.
- **Simple Installation and Maintenance.** To reduce the surface for version incompatibilities and provide an uncomplicated installation, it is highly beneficial to keep the number of software dependencies small.
  Design Decision: For installing dSalmon, we only require NumPy [14]. While dSalmon uses SWIG [6] for generating Python wrapper code and makes intensive use of Boost [13], the permissive licenses of SWIG and Boost allow us to ship any code required for compilation together with dSalmon.

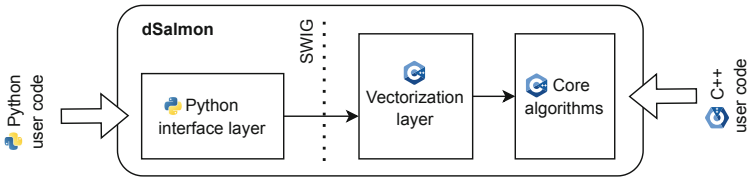


Fig. 2. Architecture of dSalmon.

Besides the above goals, it is also worth elaborating on some tasks that we explicitly constrain in the development of dSalmon.

- **Algorithm Fidelity.** A clear non-goal is the optimization of any of the implemented algorithms for OD accuracy. Such improvements inherently depend on the specific problem under investigation and, hence, are difficult to be made in an objective way. Rather, we follow the descriptions of the respective algorithm authors as closely as possible, so that users of our framework can be confident to deploy an established, well-tested method for their research tasks.
- **Maximum Code Reuse.** We see the focus of our framework in filling an important gap by providing highly efficient processing for streaming data. On the other hand, for several recurring tasks of data analysis and processing, well-functioning and comprehensive tools are provided by existing frameworks like NumPy [14], scikit-learn [24] or SciPy [30], or can trivially be implemented in a fast, vectorized manner. We explicitly avoid reimplementing tools that are already provided by established software projects to keep our code base narrow and relieve the user from having to choose between competing implementations. For instance, comprehensive metrics for evaluating the quality of an obtained outlier scoring are provided by scikit-learn, like, e.g., the ROC-AUC score or the P@n score.

4.2 Architecture

Considering design decisions in the previous Sect. 4.1, it was of importance for us to allow use of dSalmon from a programming language that is known and used by data science researchers. For this reason, we targeted the Python programming language, which allows efficient and swift data science development.

Traditional data mining algorithms for static data in many cases have at least limited opportunities for batch processing. Therefore, algorithms for static data often allow efficient implementations from an interpreted language like Python directly. However, when processing a data stream the model has to be adapted for each processed point, inherently making batch processing hardly feasible, if not impossible. To provide superior processing speed while allowing use from Python directly, dSalmon therefore implements core algorithms in C++, but provides interfaces to the algorithms from both C++ and Python.

Listing 1.1. A toy example for finding the 5 most outlying points using dSalmon.

```
from dSalmon.outlier import HSTrees
import numpy as np
detector = HSTrees(window=500, n_estimators=100, n_jobs=4)
data = np.load('data.npy')
outlier_scores = detector.fit_predict(data)
print('Outliers:', np.argsort(outlier_scores)[-5:].tolist())
```

Figure 2 depicts the architecture of dSalmon. Hence, the core algorithms layer depicted in Fig. 2 is implemented in C++. We use C++ template programming for instantiating single and double precision floating point variants of all algorithms. Researchers can thus achieve a smaller memory footprint and faster processing times by falling back to single precision processing if required. Since loop iterations are fast in C++, the core algorithms C++ interface accepts individual samples instead of blocks of data.

On the other hand, looping over individual samples in Python would incur a substantial performance penalty. Hence, the C++ vectorization layer in Fig. 2 accepts blocks of streaming data and iterates over samples within each block when passing on the data to core algorithms. To account for the dynamic nature of data streams, we allow the user to use differently sized blocks or even to vary the number of passed samples at each iteration. Additionally, the vectorization layer ensures that opportunities for parallel processing are efficiently taken by, for example, executing base detectors of ensemble methods in parallel.

For generating the actual interface between Python and C++, we deploy SWIG [6], a tool for automatically generating Python bindings by parsing C/C++ header files. The benefits of deploying SWIG are that the code base of dSalmon can easily be extended, leaving the generation of boilerplate interface code to SWIG. Since SWIG supports a wide range of target languages, our approach additionally yields the possibility to create bindings for further programming languages like, for instance, R without having to rewrite core algorithms.

The Python interface layer depicted in Fig. 2 finally accepts blocks of streaming data from user code. It accomplishes the task of ensuring a clean interface familiar in the Python community. Additionally, it performs several sanity checks on the provided data blocks.

We genuinely believe that source code should be publicly available and therefore distribute dSalmon under the LGPL 3.0 license, which permits widespread use, but requests developers to keep modified versions open-source.

4.3 Using dSalmon for Outlier Detection

Listing 1.1 shows an example of performing OD with dSalmon. In this example, the rows of `data` are interpreted as sequentially arriving samples of a data stream. As alternative to the depicted listing, a user might similarly call

`fit_predict()` sequentially with blocks of consecutive rows, or even iterate over rows in `data` individually. Since data rows are iterated by dSalmon, all three approaches provide equal results. Choosing a too small block size, however, might result in substantially slower processing. As described in Sect. 4.1, block size invariance is crucial for evaluating algorithms in a realistic manner.

4.4 M-Tree Indexing

When developing algorithms for data mining, a frequent task is finding nearest neighbors in a large set of points. In literature, this requirement gave rise to the development of various indexing data structures for performing nearest neighbor and range queries efficiently. However, many indexing trees are optimized for tree construction from bulk data and do not allow removing points and inserting new points once the tree has been built. In particular, this limitation applies to the popular KDTree and BallTree data structures provided by scikit-learn [24].

dSalmon implements an M-Tree [9] spatial indexing data structure for its internal use, which allows efficient nearest neighbor and range queries in metric spaces. By using an M-Tree, dSalmon thus allows to modify the tree after it has been built. To additionally allow algorithm development from Python, we provide a Python interface for directly using an M-Tree in custom algorithms. Similar to our further implementations, we ensured that parallel processing capabilities can efficiently be made use of and allow partially parallelized tree building and fully parallelized tree querying in an uncomplicated way by simply setting respective parameters.

5 Experimental Evaluation

In what follows, we present results from an extensive experimental evaluation that we have performed to evaluate dSalmon's resource consumption. We have performed our algorithm benchmarks on desktop machines equipped with Intel i7-4770 processors, 16GiB of main memory and no configured swap space. All machines used for evaluation have an equal setup. We used CPython version 3.7.3 and Debian Buster with kernel version 4.19.0. To avoid distorted measurements, we avoided any simultaneous use of the machines and shut down background processes as far as possible. For measuring energy consumption, we used the Running Average Power Limit (RAPL) [11] feature of our Intel CPUs, and sum memory and processor power consumption. Reported execution times do not include the time needed for loading the dataset into memory.

For performing realistic benchmarks, we selected publicly available datasets representing multivariate streaming data:

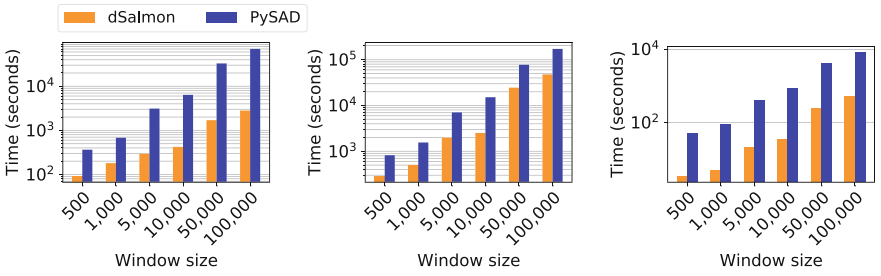
– The SWAN-SF [4] dataset provides measurement data on solar flares. To follow established preprocessing steps for SWAN-SF, we used preprocessing scripts available on the Internet [2], extracting the same features that the repository authors used in their examples. The preprocessed dataset consists

of 331,185 data samples with 12 features per sample. For assessing outlier scores, we assigned a normal label to the majority class and marked remaining classes as outliers.
- The KDD Cup'99 [1] dataset is an established dataset for OD, containing host-based and network-based features for detecting attacks in computer networks. We marked attack samples as outliers over normal traffic and used one-hot encoding for nominal features. The resulting dataset has 4,898,431 data samples with 52 features each.
- The CIC-IDS-2017 [28] dataset similarly aims at detecting network attacks, but only provides network traffic, making unsupervised attack detection substantially harder. We used an established feature vector for network traffic [32] together with publicly available preprocessing scripts [23] and consider network attacks as outliers over normal traffic. The resulting preprocessed dataset has 2,317,922 data samples and 33 features.

We selected PySAD as framework to compare it against dSalmon and performed all benchmarks using double-precision floating point processing. As described in Sect. 2, further software projects exist for OD tasks. However, the majority of these projects do not provide methods for processing streaming data. Furthermore, as well as with dSalmon, PySAD can be used from Python.

5.1 Nearest-Neighbors Algorithms

A simple approach for establishing the outlierness of arriving data points is counting the number of nearest neighbors within a pre-determined radius. Hence, in many traditional publications [3,19,34] an arriving data point is declared to be an outlier if less than $k \in \mathbb{N}$ points of the current SW lie within a radius $R \in \mathbb{R}^+$. While modern approaches for OD frequently outperform this simple nearest-neighbors-based approach in both, detection accuracy and execution time, the importance of a simple nearest-neighbors-based approach lies in its unrivaled interpretability of reported outlier scores, making its availability cru-



(a) CIC-IDS-2017 (b) KDD Cup'99 (c) SWAN-SF

Fig. 3. Execution time comparison for nearest-neighbors-based streaming outlier detection.

cial for dSalmon. Interpretability is crucial in various fields of application like, e.g., medicine [20,31], or network intrusion detection [5,15,21].

While providing a binary label (inlier/outlier) in some cases is sufficient in practice, for research and parameter selection it is usually necessary to obtain a score for outlierness for data samples. When requiring an outlier score instead of a binary label, distance-based OD can be performed in two flavors:

1. When implementing a SW-based k nearest neighbors (kNN) rule, OD can be parameterized by the neighbor count k, providing the distance to the k^{th} nearest point as outlier score.
2. Alternatively, OD can be parameterized by the search radius R, providing the number of neighbors within R as inverse outlier score.

dSalmon allows OD using both flavors (1) and (2), termed SW-KNN and SW-DBOR, respectively, and deploys M-Tree indexing to reduce execution time.

PySAD supports nearest-neighbors-based OD only in flavor (2), adopting the name ExactStorm from [3] for this model. We thus used this operational mode also for dSalmon, resulting in execution times as shown in Fig. 3 for different lengths of the SW. To provide a meaningful comparison, for each individual window size we used grid search on a logarithmic scale for finding the radius R that optimizes the ROC-AUC score when applying the algorithm to the complete dataset, and used the resulting R for performing the benchmark.

Execution time benefits demonstrated in Fig. 3 can be explained by two effects: On the one hand, for small window lengths execution time is dominated by interpretation overhead of the Python language for PySAD, which dSalmon avoids due to its C++ core implementation. However, PySAD sensibly performs distance computations for each processed data sample in a vectorized manner, diminishing the interpretation overhead as the window length increases. As shown in Fig. 3, dSalmon is able to retain a substantial speedup even as window size increases. This observation demonstrates execution time benefits of M-Tree indexing compared to straight per-sample distance computations.

5.2 Ensemble-Based Outlier Detectors

In recent research, an increasingly popular approach for OD, which sets new records in detection accuracy, is to construct algorithms by pooling outlier scores obtained by an ensemble of weak learners. Beneath yielding good accuracy, this approach is intrinsically embarrassingly parallel, as the processing of distinct base detectors can trivially be distributed to several workers. dSalmon allows to easily leverage this feature by simply setting an `n_jobs` parameter.

When evaluating execution performance, for the sake of providing a fair comparison, we chose algorithms whose specification leaves little room for interpretation. In particular, we selected the following methods:

[1] Missing results for RRCF using PySAD indicate experiment runs that failed due to reaching Python's recursion limit.

[2] xStream for KDD Cup'99 using PySAD with 50 random projections failed due to running out of memory.

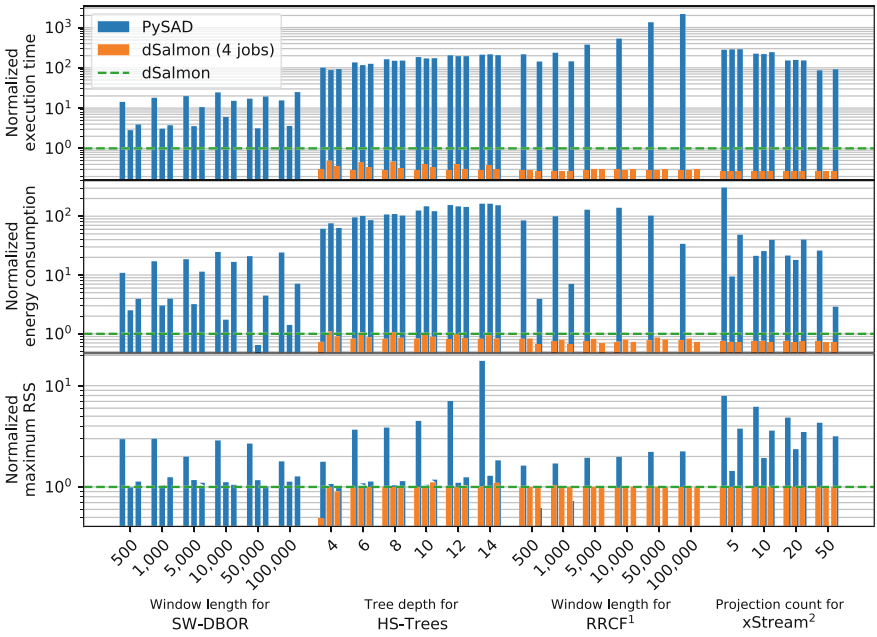
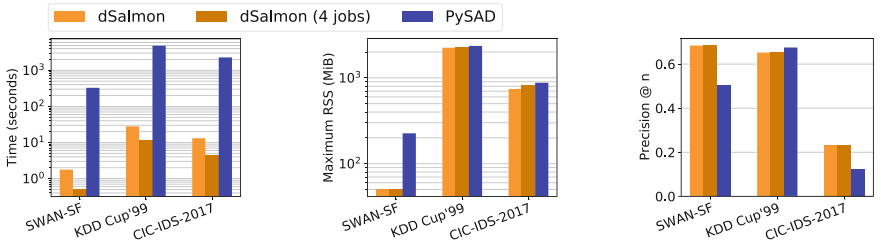


Fig. 4. Overall comparison of the resource consumption of several outlier detectors implemented by dSalmon and PySAD. For each parameter setting, values are normalized to results obtained by single-threaded dSalmon for better comparison. Bars depicted for each algorithm parameterization indicate results for SWAN-SF, KDD Cup'99 and CIC-IDS-2017 in this order.

- Robust Random Cut Forest (RRCF) [12] uses an ensemble of dynamically constructed trees, where each tree performs random cuts based on the feature space of observed samples. Concept drift is taken care of using a SW approach. We perform runs with varying window sizes to show dependence when varying this parameter.
- Half-Space-Trees [29] similarly constructs a tree ensemble, but performs tree construction statically at the time of algorithm initialization. Concept drift is considered based on a RW approach. In our experiments, we vary the depth of the constructed trees to evaluate influence of tree depth on resource usage.
- xStream [22] is a recent method for OD, which introduces half-space-chains, which establish an anomaly score by splitting randomly selected features with varying precisions. xStream combines half-space-chains with the technique of random projections. For benchmarking, we set the chain length to 15 as used for the evaluations in [22] and vary the number of projections to show dependence on this parameter.

In our experiments, we use an ensemble size of 100 for all algorithms, which is similarly used as default value for ensemble methods by scikit-learn. Using 100 base estimators as ensemble size is a natural choice and is likely to reduce



(a) Time (b) Memory (c) P@n

Fig. 5. HS-Trees: Resource consumption and OD performance using a tree depth of 10.

statistical variation of outlier scores to an acceptable level. For dSalmon, we evaluate performance for both single-threaded operation and when utilizing four processor cores. Multi-threaded operation in not supported by PySAD.

In Fig. 4 we depict a summarized comparison of resource consumption that we measured while performing OD with dSalmon and PySAD on the SWAN-SF, KDD Cup'99 and CIC-IDS-2017 datasets, also including the results already presented in Sect. 5.1. We depict execution time, memory usage as maximum Resident Set Size (RSS) and energy consumption. Since we aim to depict relative performance when using both frameworks, we normalize all measurements by results obtained when using dSalmon on one processor core.

Figure 4 shows that, particularly for modern ensemble-based algorithms, dSalmon yields substantial execution time benefits. For most algorithm runs, a speed-up by a factor of more than 100 can be obtained. From Fig. 4 we can additionally conclude that dSalmon makes highly efficient use of parallel processing capabilities. Execution time indicates that, by using four simultaneous jobs, a speed-up of almost 4 can be obtained in most cases. Furthermore, memory consumption does not increase when relying on parallel processing, allowing highly efficient operation on modern multi-core desktop machines or servers. It is also interesting to note that by relying on multi-threaded processing considerable energy savings can be obtained.

In what follows, we will analyze behavior for specific algorithms in more detail. We skip RRCF for closer analysis, as for RRCF many PySAD runs failed due to reaching the maximum recursion limit, which cannot be fixed safely [33]. RRCF results that we obtained for SWAN-SF indicate that dSalmon shows markedly low runtimes, especially if longer memory lengths are required.

Half-Space-Trees Figure 5 shows the absolute measurement readings that we obtained for HS-Trees with a tree depth of 10. In the light of execution times in Fig. 5 (a), we can generally attest HS-Trees' outstanding run time performances, since HS-Trees is able to process millions of data samples in about 10 s.

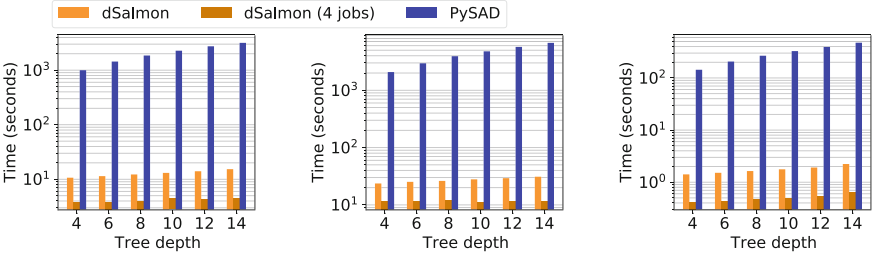
It is worth noting that, besides obtaining markedly different execution times, we also obtained deviating results for detection performance when deploying

both implementations, as shown in Fig. 5(c). This can be explained by PySAD basing the reported outlier score on all tree nodes traversed for one sample, while dSalmon computes outlier scores only from terminal nodes as suggested by Tan et al. [29] when introducing HS-Trees. Figure 5(c) additionally demonstrates reproducibility of obtained results for dSalmon. Hence, for both independent algorithm runs – one utilizing one CPU core and another utilizing four cores – the precise same outlier scores are reported, since the same seed value has been provided as parameter. This holds true even though both runs differ in their parameterization for parallel processing. We have verified this property also for all further runs depicted in Fig. 4.

In Fig. 6, we depict absolute execution times as function of tree depth. Hence, dSalmon obtains an approximately 100 times speed-up and execution time shows only a slight increase when increasing the tree depth. In fact, rather memory occupation is limiting the usable maximum tree depth, since for HS-Trees tree structures are statically created, resulting in memory consumption that increases exponentially with tree depth.

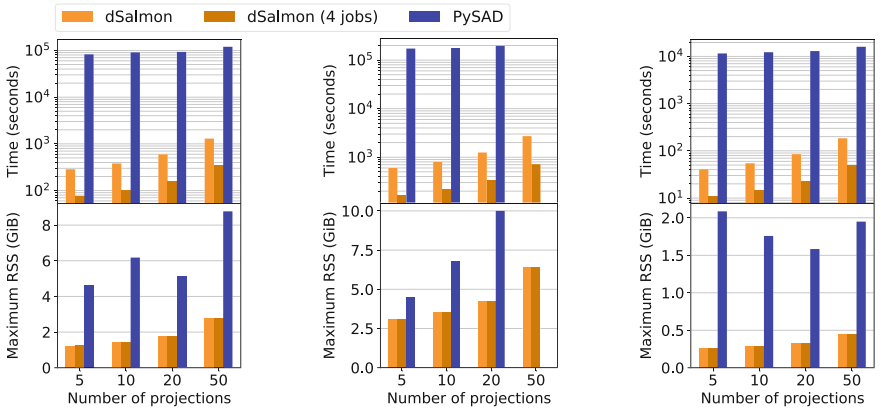
xStream Figure 7 shows absolute execution times und maximum RSS as function of the number of projections for xStream. As discussed by the algorithm authors [22], execution time depends linearly on ensemble size, chain length and the number of projections. Hence, observed behavior for our dSalmon runs is reasonable. We notice that PySAD shows a less pronounced dependency of the number of projections. Consequently, execution time differences of dSalmon range between 20 and 200. dSalmon proves to make highly efficient use of parallel processing, allowing a 4-times speed-up by using 4 parallel jobs.

Memory consumption as a function of the number of projections, as depicted in Fig. 7, is particularly striking. While PySAD's memory consumption shows no clear dependence of the number of projections for SWAN-SF, for KDD Cup'99 a clear monotonic dependence can be observed, and the algorithm run eventually fails on our 16GiB machine due to running out of memory for 50 projections. xStream authors [22] suggest using a Count-min sketch (CMS) [10] for counting bin frequencies to ensure constant space complexity. In dSalmon, we adopt the



(a) CIC-IDS-2017 (b) KDD Cup'99 (c) SWAN-SF

Fig. 6. Runtimes of HS-Trees in response to variations of the tree depth.



(a) CIC-IDS-2017 (b) KDD Cup'99 (c) SWAN-SF

Fig. 7. xStream: Resource consumption in response to variations of the number of projections.

approach of using CMSs, while PySAD uses classical hash tables. The use of hash tables for this purpose provides an explanation for the data-dependent memory consumption observable in Fig. 7, since memory consumption in this case is reduced if the majority of data samples shares a small number of bins. In dSalmon, memory requirements for storing CMS structures are independent of the projection count. The increase of memory consumption can be explained by the memory requirements for storing projected values for a given block of data.

6 Conclusions

We have introduced and presented dSalmon, a highly efficient framework for OD on multivariate evolving data streams. Due to the nature of streaming data, data samples frequently accumulate to a substantial volume within short time, making efficient processing crucial. We have presented dSalmon's architecture, which allows easy extension and enables researchers and practitioners to add algorithms for OD or even implementing entirely different methods for analyzing streaming data.

In a thorough evaluation, we have shown that dSalmon was able to outperform existing Python stream outlier detectors by up to three orders of magnitude with respect to execution time. Combined with the selection of a recent OD method optimized for processing high-rate data streams, gigabytes of data can be processed in few seconds, paving the way for analyzing comprehensive datasets, which increasingly become available due to advances of modern technology.

References

1. Kdd cup 1999 data. https://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html (1999), accessed: 2023–07-04
2. Ahmadzadeh, A., Aydin, B.: Multivariate Timeseries Feature Extraction on SWAN Data Benchmark (SWAN_Features) (2020), GSU Data Mining Lab
3. Angiulli, F., Fassetti, F.: Detecting distance-based outliers in streams of data. In: Proceedings of the 16th ACM Conference on Information and Knowledge Management, pp. 811–820, CIKM'07, ACM, New York, NY, USA (2007)
4. Angryk, R.A., Martens, P.C., Aydin, B., Kempton, D., Mahajan, S.S., Basodi, S., Ahmadzadeh, A., Cai, X., Filali Boubrahimi, S., Hamdi, S.M., Schuh, M.A., Georgoulis, M.K.: Multivariate time series dataset for space weather data analytics. Sci. Data **7**(227) (2020)
5. Bachl, M., Hartl, A., Fabini, J., Zseby, T.: Walling up backdoors in intrusion detection systems. In: Big-DAMA '19, pp. 8–13. ACM, Orlando, FL, USA (2019)
6. Beazley, D.M.: SWIG: An easy to use tool for integrating scripting languages with C and C++. In: Proceedings of the 4th Conference on USENIX Tcl/Tk Workshop, 1996 - Volume 4, p. 15, TCLTK'96, USENIX Association, USA (1996)
7. Bifet, A., Holmes, G., Kirkby, R., Pfahringer, B.: Moa: massive online analysis. J. Mach. Learn. Res. **11**, 1601–1604 (2010)
8. Campos, G.O., Zimek, A., et al.: On the evaluation of unsupervised outlier detection: measures, datasets, and an empirical study. Data Mining and Knowl. Discovery **30**(4), 891–927 (2016), ISSN 1573–756X
9. Ciaccia, P., Patella, M., Zezula, P.: M-tree: An efficient access method for similarity search in metric spaces. In: Proceedings of the 23rd International Conference on Very Large Data Bases, pp. 426–435, VLDB '97, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1997), ISBN 1558604707
10. Cormode, G., Muthukrishnan, S.: An improved data stream summary: the count-min sketch and its applications. J. Algorithms **55**(1), 58–75 (2005)
11. David, H., Gorbatov, E., Hanebutte, U.R., Khanna, R., Le, C.: Rapl: memory power estimation and capping. In: 2010 ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED). IEEE, pp. 189–194 (201)
12. Guha, S., Mishra, N., Roy, G., Schrijvers, O.: Robust random cut forest based anomaly detection on streams. In: Proceedings of The 33rd International Conference on Machine Learning, Proceedings of Machine Learning Research, vol. 48, pp. 2712–2721, PMLR, New York, New York, USA (2016)
13. Gurtovoy, A., Abrahams, D.: The boost C++ metaprogramming library, p. 22 (2002)
14. Harris, C.R., Millman, K.J., van der Walt, S.J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N.J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M.H., Brett, M., Haldane, A., del Río, J.F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., Oliphant, T.E.: Array programming with NumPy. Nature **585**(7825), 357–362 (2020)
15. Hartl, A., Bachl, M., Fabini, J., Zseby, T.: Explainability and adversarial robustness for RNNs. In: 2020 IEEE Sixth International Conference on Big Data Computing Service and Applications (BigDataService), pp. 148–156. IEEE, New York, NY, USA (2020a)
16. Hartl, A., Iglesias, F., Zseby, T.: SDOstream: Low-density models for streaming outlier detection. In: ESANN 2020 proceedings, pp. 661–666 (2020b)

17. Iglesias, F., Hartl, A., Zseby, T., Zimek, A.: Are network attacks outliers? a study of space representations and unsupervised algorithms. In: Joint European Conference on Machine Learning and Knowledge Discovery in Databases, pp. 159–175. Springer (2019)
18. Iglesias Vázquez, F., Hartl, A., Zseby, T., Zimek, A.: Anomaly detection in streaming data: A comparison and evaluation study. Expert Syst. with Appl. **233**, 120994 (2023), ISSN 0957–4174
19. Kontaki, M., Gounaris, A., Papadopoulos, A.N., Tsichlas, K., Manolopoulos, Y.: Continuous monitoring of distance-based outliers over data streams. In: IEEE 27th International Conference on Data Engineering, pp. 135–146 (2011)
20. Lakkaraju, H., Rudin, C.: Learning cost-effective and interpretable treatment regimes. In: Proceedings of the 20th International Conference on Artificial Intelligence and Statistics, pp. 166–175, PMLR, Fort Lauderdale, FL, USA (2017)
21. Lundberg, H., Mowla, N.I., Abedin, S.F., Thar, K., Mahmood, A., Gidlund, M., Raza, S.: Experimental analysis of trustworthy in-vehicle intrusion detection system using explainable artificial intelligence (xai). IEEE Access **10**, 102831–102841 (2022)
22. Manzoor, E.A., Lamba, H., Akoglu, L.: xStream: outlier detection in feature-evolving data streams. In: 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (2018)
23. Meghdouri, F.: Datasets Preprocessing (2021). https://github.com/CN-TU/Datasets-preprocessing, gitHub repository
24. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E.: Scikit-learn: machine learning in Python. J. Mach. Learn. Res. **12**, 2825–2830 (2011)
25. Pevný, T.: Loda: Lightweight on-line detector of anomalies. Mach. Learn. **102**(2), 275–304 (2016)
26. Sathe, S., Aggarwal, C.C.: Subspace outlier detection in linear time with randomized hashing. In: 2016 IEEE 16th International Conference on Data Mining (ICDM), pp. 459–468 (2016)
27. Schubert, E., Zimek, A.: Elki: A large open-source library for data analysis—elki release 0.7.5 "heidelberg". arXiv preprint arXiv:1902.03616 (2019)
28. Sharafaldin, I., Habibi Lashkari, A., Ghorbani, A.A.: Toward generating a new intrusion detection dataset and intrusion traffic characterization. In: ICISSP, pp. 108–116, SCITEPRESS, Funchal, Madeira, Portugal (2018)
29. Tan, S.C., Ting, K.M., Liu, T.F.: Fast anomaly detection for streaming data. In: Twenty-Second International Joint Conference on Artificial Intelligence (2011)
30. Virtanen, P., Gommers, R., Oliphant, T.E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S.J., Brett, M., Wilson, J., Millman, K.J., Mayorov, N., Nelson, A.R.J., Jones, E., Kern, R., Larson, E., Carey, C.J., Polat, İ., Feng, Y., Moore, E.W., VanderPlas, J., Laxalde, D., Perktold, J., Cimrman, R., Henriksen, I., Quintero, E.A., Harris, C.R., Archibald, A.M., Ribeiro, A.H., Pedregosa, F., van Mulbregt, P., SciPy 1.0 contributors: SciPy 1.0: fundamental algorithms for scientific computing in Python. Nature Methods **17**, 261–272 (2020)
31. Weng, S.F., Reps, J., Kai, J., Garibaldi, J.M., Qureshi, N.: Can machine-learning improve cardiovascular risk prediction using routine clinical data? PLoS ONE **12**(4), 1–14 (2017)

32. Williams, N., Zander, S., Armitage, G.: A preliminary performance comparison of five machine learning algorithms for practical IP traffic flow classification. SIGCOMM Comput. Commun. Rev. **36**(5), 5–16 (2006)
33. Wouters, T.: Answer to "what is the maximum recursion depth in python, and how to increase it?" (2010). https://stackoverflow.com/a/3323013, stackoverflow discussion
34. Yang, D., Rundensteiner, E., Ward, M.O.: Neighbor-based pattern detection for windows over streaming data. In: Proceedings of the 12th International Conference on Extending Database Tech.: Advances in Database Tech., pp. 529–540, EDBT'09, ACM, New York, NY, USA (2009)
35. Yilmaz, S.F., Kozat, S.S.: Pysad: a streaming anomaly detection framework in Python (2020). arXiv preprint arXiv:2009.02572
36. Zhao, Y., Nasrullah, Z., Li, Z.: Pyod: a python toolbox for scalable outlier detection. J. Mach. Learn. Res. **20**(96), 1–7 (2019)