OutlierDetection.jl: A modular outlier detection ecosystem for the Julia programming language

David Muhr¹,²
Michael Affenzeller²,³
Anthony D. Blaom⁴

¹ BMW Group, Steyr, Austria.
² Institute for Formal Models and Verification, Johannes Kepler University Linz, Austria.
³ HEAL, University of Applied Sciences Upper Austria, Austria.
⁴ University of Auckland, New Zealand.

Abstract

OutlierDetection.jl is an open-source ecosystem for outlier detection in Julia. It provides a range of high-performance outlier detection algorithms implemented directly in Julia. In contrast to previous packages, our ecosystem enables the development highly-scalable outlier detection algorithms using a high-level programming language. Additionally, it provides a standardized, yet flexible, interface for future outlier detection algorithms and allows for model composition unseen in previous packages. Best practices such as unit testing, continuous integration, and code coverage reporting are enforced across the ecosystem. The most recent version of OutlierDetection.jl is available at https://github.com/OutlierDetectionJL/OutlierDetection.jl.

Keywords: outlier detection, anomaly detection, machine learning, Julia

1. Introduction

An outlier is "an observation (or subset of observations) which appears to be inconsistent with the remainder of that set of data" (Barnett and Lewis, 1978). Outlier detection is the research area that studies the detection of such inconsistent observations. Outliers are by nature infrequent events, thus labels are difficult to obtain and the ground truth is often absent in outlier detection tasks. Outlier detection is mainly used in fields that process large amounts of unlabelled data, such as network intrusion detection, fraud detection, medical diagnostics or industrial quality control. For reviews covering the most important outlier detection application areas refer to Chandola et al. (2009) and Pimentel et al. (2014).

Software packages for outlier detection exist in various programming languages such as ELKI Data Mining (Achtert et al., 2010) in Java, DDoutlier (Madsen, 2018) in R, or PyOD (Zhao et al., 2019) in Python. Existing outlier detection packages, however, either cater to a research community, where benchmark datasets are small and performance is negligible, or are plagued by the "two language problem" to define algorithms. The two language problem refers to the necessary usage of multiple programming languages to achieve high-performing numerical code, e.g., Python and NumPy, calling C functions under the hood Bezanson et al.
(2017). *OutlierDetection.jl* aims to address these problems, allowing future researchers to define scalable algorithms using the Julia programming language.

## 2. Design Goals

The following points summarize the key design goals we followed in the development of *OutlierDetection.jl*. Most importantly, we choose to contribute to Julia’s existing machine learning community instead of building yet another separate machine learning package.

**Community.** An open-source community has been founded to facilitate the collaboration of ecosystem contributors. Additionally, tight integration to the rest of Julia’s machine learning ecosystem is provided, integrating with the most common machine learning libraries and directly extending Machine Learning in Julia (MLJ) (Blaom et al., 2020).

**Modularity.** The functionality is split into multiple packages, and each package fulfills a specific purpose, see Figure 1 for an overview. A modular package structure has shown to be beneficial in other large Julia projects such as *POMDPs.jl* (Egorov et al., 2017) or *MLJ* (Blaom et al., 2020). On the one hand, this modularity enables developers to contribute more easily, and, on the other hand, the complexity is hidden from the end-users, which have a single point of entry that does not load all dependencies upfront, which is especially important with Julia’s precompilation.

**Quality.** Unit testing and code coverage reporting is used to test the internal procedures of all packages in the ecosystem. Continuous integration is used to conduct automated testing under various versions of Julia and operating systems. After each commit, or when a pull request is opened, automatic cross-platform tests are executed to ensure that the code quality meets our standards.

**Documentation.** Documentation is developed in a unified notion for the developers and users of *OutlierDetection.jl* by standardizing the comment structure and directly transforming comments to user-readable documentation of the entire API using *Documenter.jl*. Additional documentation such as usage examples are written in markdown with executable code blocks, ensuring that the documentation does not get out of sync with the codebase.

**Relevance.** *OutlierDetection.jl* is the first major outlier detection project for the Julia programming language. It is the first project that tackles the two-language problem in outlier detection, enabling researchers to develop high-performing outlier detection algorithms that scale to millions of data points with comparably little effort.

**Standardization.** Typically, each novel outlier detection algorithm comes with its own set of assumptions, for example, that the outlier class is defined as 1 or -1. Using a new concept called *scientific types* (Király et al., 2021), we can standardize such decisions without limiting the algorithm authors’ flexibility in their machine-type representation.

---

1. https://julialang.org
2. https://github.com/OutlierDetectionJL
3. https://github.com/JuliaDocs/Documenter.jl
3. Implementation

As mentioned previously, one of the goals of this library is to integrate with the rest of Julia’s machine learning ecosystem. Julia’s multiple dispatch (Bezanson et al., 2017) enables us to augment other libraries and compose functionality across packages and organizations. One example of this composability is the integration with MLJ, which we use as an entry point for users to discover outlier detection algorithms.

![OutlierDetection.jl package overview and MLJ integration](image)

Figure 1: OutlierDetection.jl package overview and MLJ integration

In Figure 1, we show the conceptual architecture of the packages contained in the outlier detection organization and how they interoperate with MLJ. All outlier detection models are directly usable in MLJ, without knowing any of the underlying package names beforehand. This is possible because all models are added to MLJ’s model registry. Listing all unsupervised outlier detection models is shown in Figure 2.

```julia
using MLJ

function task(model)
    model.abstract_type == UnsupervisedDetector

function models(task) # task is a function to filter existing models

```

Figure 2: Listing all unsupervised outlier detection models.

One of the major design decisions is to separate the tasks occurring in outlier detection into separate packages. Outlier detection algorithms typically assign an outlier score to each instance in the dataset (Aggarwal, 2017), as visible in Figure 3.

The conversion of outlier scores to labels or probabilities is a separate task usually based on a threshold identified from the training data. Because scoring and score conversion are mostly separate tasks, we centralize various score conversion utilities in the OutlierDetection.jl package. Therefore, developers of new detection algorithms do not have to implement score conversion utilities themselves but are free to implement their utilities if required. We provide multiple ways to convert scores; one of the approaches is using a model wrapper as shown in Figure 4. There is one more user-facing package, not yet mentioned in the examples, namely OutlierDetectionData.jl. This package provides a unified interface to load common outlier detection benchmark datasets as shown in Figure 5.
using MLJ, OutlierDetection
KNN = @load KNNDetector pkg = OutlierDetectionNeighbors
X = rand(10, 1000); # generate 10-dimensional points
model = KNN(); # model: collection of hyperparameters
mach = machine(model, X); # machine: model bound to data
fit!(mach) |> transform # fit!/transform: learn a model and assign scores

Figure 3: Transform a number of points into raw outlier scores.

using MLJ, OutlierDetection
KNN = @load KNNDetector pkg = OutlierDetectionNeighbors
X = rand(10, 1000); # generate 10-dimensional points
model = DeterministicDetector(KNN()); # wrapped model adding `predict`
machine(model, X) |> fit! |> predict # predict labels from scores

Figure 4: Predict labels instead from the raw outlier scores.

using OutlierDetectionData: ODDS
ODDS.list() # list all datasets in the collection
X, y = ODDS.load("annthyroid") # load a specific dataset

Figure 5: Load a dataset from the ODDS collection (Rayana, 2016).

The integration with MLJ enables sophisticated use cases to be defined in a declarative fashion. For example, hyperparameter tuning, model evaluation, ensemble learning, or complex model composition in terms of learning networks can be achieved (Blaom and Vollmer, 2020). Ensemble techniques have shown to be beneficial in many outlier detection tasks (Aggarwal and Sathe, 2015) and model composition is recently investigated, for example combining deep self-supervised representation learning with nearest-neighbor search (Bergman et al., 2020).

4. Conclusion

This paper presents an outlier detection ecosystem for Julia that allows researchers to implement scalable outlier detection algorithms in a high-level programming language. A carefully designed library interface enables model composition unseen in previous outlier detection packages. One key factor enabling model composition is the clear separation of scoring and score conversion of outlier detection models. We believe that OutlierDetection.jl will allow future researchers to quickly iterate on new outlier detection algorithms, and model composition will enable novel outlier detection use cases and methods.
References

Elke Achtert et al. Visual Evaluation of Outlier Detection Models. In *Database Systems for Advanced Applications*, volume 5982, pages 396–399. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. ISBN 978-3-642-12097-8 978-3-642-12098-5. doi: 10.1007/978-3-642-12098-5_34.

Charu C. Aggarwal. *Outlier Analysis*. Springer International Publishing, Cham, January 2017. ISBN 978-3-319-47577-6. doi: 10.1007/978-3-319-47578-3.

Charu C. Aggarwal and Saket Sathe. Theoretical Foundations and Algorithms for Outlier Ensembles. *ACM SIGKDD Explorations Newsletter*, 17(1):24–47, January 2015. ISSN 1931-0145. doi: 10.1145/2830544.2830549.

Vic Barnett and Toby Lewis. *Outliers in Statistical Data*. John Wiley & Sons, Inc, January 1978. ISBN 978-0-471-99599-9.

Liron Bergman, Niv Cohen, and Yedid Hoshen. Deep nearest neighbor anomaly detection. *CoRR*, abs/2002.10445, 2020.

Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A Fresh Approach to Numerical Computing. *SIAM Review*, 59(1):65–98, January 2017. ISSN 0036-1445, 1095-7200. doi: 10.1137/141000671.

Anthony Blaom, Franz Kiraly, Thibaut Lienart, Yiannis Simillides, Diego Arenas, and Sebastian Vollmer. MLJ: A Julia package for composable machine learning. *Journal of Open Source Software*, 5(55):2704, November 2020. ISSN 2475-9066. doi: 10.21105/joss.02704.

Anthony D. Blaom and Sebastian J. Vollmer. Flexible model composition in machine learning and its implementation in MLJ. *arXiv:2012.15505 [cs]*, December 2020.

Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection. *ACM Computing Surveys*, 41(3):1–58, January 2009. ISSN 03600300. doi: 10.1145/1541880.1541882.

Maxim Egorov, Zachary N. Sunberg, Edward Balaban, Tim A. Wheeler, Jayesh K. Gupta, and Mykel J. Kochenderfer. POMDPs.jl: A framework for sequential decision making under uncertainty. *Journal of Machine Learning Research*, 18(26):1–5, 2017.

Franz J Király, Markus Löning, Anthony Blaom, Ahmed Guercioeur, and Raphael Sonabend. Designing Machine Learning Toolboxes: Concepts, Principles and Patterns. page 35, 2021.

Jacob H. Madsen. DDoutlier: Distance & Density-Based Outlier Detection, May 2018.

Marco A.F. Pimentel, David A. Clifton, Lei Clifton, and Lionel Tarassenko. A review of novelty detection. *Signal Processing*, 99:215–249, January 2014. ISSN 01651684. doi: 10.1016/j.sigpro.2013.12.026.

Shebuti Rayana. ODDS Library. http://odds.cs.stonybrook.edu, 2016.

Yue Zhao, Zain Nasrullah, and Zheng Li. PyOD: A python toolbox for scalable outlier detection. *Journal of Machine Learning Research*, 20(96):1–7, 2019.