



Review of PySR: high-performance symbolic regression in Python and Julia

Alberto Tonda¹

Published online: 23 December 2024

© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2024

PySR 0.19.4 is an open-source, freely available Python module for Symbolic Regression (SR) written and maintained by Dr. Miles Cranmer of the University of Cambridge [1]. The module relies upon an internal engine written in Julia (**SymbolicRegression.jl**), optimized for computational efficiency, which can also be employed separately in Julia projects. PySR has excellent documentation and is under active development.

The main features that PySR offers to data scientists and practitioners are fully scikit-learn [2] compatible classes for a symbolic regression based regressor (**PySRRRegressor**) and a classifier (**PySRClassifier**), with scikit-learn being the de facto standard for machine learning algorithms in Python. PySRRRegressor and PySRClassifier can be thus seamlessly integrated into any Python machine learning pipeline testing different algorithms of the same kind. E.g. a PySRRRegressor object could replace a scikit-learn RandomForestRegressor or RidgeRegressor, as they share the same API. Both the regressor and classifier classes offer similar options, allowing users to specify population size, number of generations, building blocks for the symbolic regression trees, and so on. PySR is also perfectly integrated with the most commonly used packages in data science, for example with the possibility of obtaining the final equations as a pandas¹ DataFrame, or sympy's symbolic expressions², or even as LaTeX code (Fig. 1).

To Genetic Programming people, PySR offers a computationally efficient symbolic regression implementation with recent and sophisticated symbolic regression features, and a code base that can be relatively easily modified to test new ideas.

PySR parallelizes function evaluations (using multiprocessing by default, but multithreading is also available), and can easily exploit more computing power by deploying on multi-node clusters. In fact, by default PySR uses a multi-population

¹ pandas is arguably the most used Python/R library for manipulating tables and CSV files, https://pandas.pydata.org/docs/user_guide/index.html

² sympy is a popular library for symbolic mathematics in Python, <https://www.sympy.org/en/index.html>

✉ Alberto Tonda
alberto.tonda@inrae.fr

¹ Applied Mathematics and Computer Science, Paris Saclay Joint Research Unit (UMR 518 MIA-PS), INRAE, Université Paris-Saclay, Palaiseau, France

```

import numpy as np
from pysr import PySRRRegressor

# generate 5 random features, and a one-dimensional target y,
# which in reality is function of only two of the features (x3, x0)
X = 2 * np.random.randn(100, 5)
y = 2.5382 * np.cos(X[:, 3]) + X[:, 0] ** 2 - 0.5

# instantiate a PySRRRegressor with custom options
model = PySRRRegressor(
    populations=5,
    population_size=100,
    niterations=40, # < Increase me for better results
    binary_operators=["+", "*"],
    unary_operators=[
        "cos",
        "exp",
        "sin",
        "inv(x) = 1/x",
        # ^ Custom operator (julia syntax)
    ],
    extra_sympy_mappings={"inv": lambda x: 1 / x},
    # ^ Define operator for SymPy as well
)

# fit model
model.fit(X, y)
# print equation that has been heuristically selected as the best
print(model.sympy())
# save all equations on Pareto front to CSV (equations_ is a pandas DataFrame)
model.equations_.to_csv("all_equations.csv")

```

Fig. 1 PySRRRegressor trying to fit $y=2.5382\cos(x_3)+x_0^2-0.5$ for 100 random values of x , using five populations each of 100 individuals and running for 40 generations. Notice the custom user-defined unary operator *inv*. The Pareto best evolved expressions are stored in the *equations_* attribute of the PySRRRegressor object, and then saved to the file *all_equations.csv*

scheme with separate islands, each one associated with a different core. The number of populations is a parameter of each PySR class. In the documentation, Cranmer offers practical advice for tuning the algorithm and improving speed.³

At the end of each evolution, PySR does not return a single solution, but rather a set of candidate equations, each one a compromise between complexity and fitting. Complexity can be tuned by defining weights for each element appearing in a tree, via the *complexity_of_operators* argument. PySR also features a variety of useful options for practical applications, such as applying denoising to the input data, performing feature selection or Principal Component Analysis before starting the symbolic regression process, and inserting expert knowledge into the expressions by seeding or adding constraints, which may limit nesting of expressions (e.g.

³ Advice for improving PySR speed in the documentation, <https://astroautomata.com/PySR/tuning/>

to avoid poorly interpretable sequences of `sin(cos(sin(sin(...)))` or even overall frequency of appearance of specific building blocks. PySR also offers the possibility of adding personalized building blocks to symbolic regression or replacing the fitness function, defining personalized functions in Julia. Despite all possibilities of customization, the default settings for PySRRRegressor deliver good results for most applications, making it usable even by non specialists.

While I consider PySR a remarkable resource for ease of use, customizability, and speed, it has its drawbacks. For example, its computational efficiency comes at the price of reproducibility, as fixing the random seed for the pseudo-random number generators has no effect when multi-processing is active. Keeping pseudo-random number generation coherent across parallel processes is a complex software engineering issue, and currently the only way of ensuring reproducibility is by forcing PySR to run on a single process, which of course impairs performance. Furthermore, while introducing new operators and fitness functions is not difficult, it still requires defining them using Julia syntax, which from the Python interface corresponds to writing Julia code inside Python strings; this can make debugging quite intricate. Finally, a minor issue for practitioners is that the heuristic used to automatically select a candidate equation on the final front of compromises can be flawed, and sometimes ignores better candidates; nevertheless, all equations on the front are saved and can be accessed individually.

Despite some small drawbacks, PySR remains one of the few examples of modern, usable, computationally-efficient symbolic regression I could find. The only comparable implementation of symbolic regression I am aware of is Operon [3], developed in C++, which also features scikit-learn compatible bindings for Python.⁴ Being compiled in C++, Operon is much faster (about 0.04 s versus PySR's 7 s to evaluate 100,000 candidates on my Linux laptop); the ease of use is similar, as both projects have a Python interface; however, modifying PySR for different purposes (e.g. testing research ideas) seems easier, as at worst PySR requires tweaking with Julia code, which is relatively high-level when compared to Operon's C++. PySR documentation is also more extensive and informative, and the current version of Operon 0.4.0 still has a few issues. For example, when attempting to install Operon with Anaconda under Microsoft Windows, I faced several C++ compilation errors, while a Linux installation had no problems. It is worth mentioning that both Operon and PySR are part of the algorithms currently being tested in the SRBench benchmarking effort.⁵

PySR can be easily installed through pip (`sudo pip install pysr`), or cloned from its GitHub repository,⁶ but I would recommend pip, as installing it from GitHub currently also requires manually installing Julia. For more information, consult the documentation.⁷ On Microsoft Windows, PySR can be installed through Anaconda or Anaconda Cloud distributions.

⁴ PyOperon GitHub repository, <https://github.com/heal-research/pyoperon>

⁵ SRBench GitHub repository, <https://github.com/cavalab/srbench>

⁶ PySR GitHub repository, <https://github.com/MilesCranmer/PySR>

⁷ PySR documentation, <https://astroautomata.com/PySR/>

References

1. M. Cranmer, Interpretable Machine Learning for Science with PySR and SymbolicRegression.jl (2023). arXiv: <https://arxiv.org/abs/2305.01582>
2. F. Pedregosa et al., Scikit-learn: machine learning in python. JMLR 12, pp. 2825–2830 (2011). <https://jmlr.csail.mit.edu/papers/v12/pedregosa11a.html>
3. B. Burlacu et al. Operon C++: an efficient genetic programming framework for symbolic regression. In: Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion, pp. 1562–1570 (2020). <https://doi.org/10.1145/3377929.3398099>

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.