Research Internship Report

Bayesian Symbolic Regression

David Coba

St. no. 12439665

Research Master's Psychology

University of Amsterdam

Psychological Methods

Supervised by:

Don van den Bergh

Eric-Jan Wagenmakers

XX August 2021

Abstract

Symbolic regression is a supervised machine learning method that outputs mathematical expressions. Therefore the models are fully interpretable, unlike other machine learning techniques. This allows symbolic regression to act as a bridge between black-box machine learning models and scientific models that are represented mathematically.

This project aimed to develop and test a Bayesian symbolic regression algorithm. We build from the model of Jin et al. (2020) and introduce two significant modifications: a symbolic simplification step and the possibility of running multiple Markov chain Monte Carlo chains in parallel. The symbolic simplification step model performed worse than the original algorithm, but the multi-chain sampler was more performant. Moreover, Bayesian models discovered similar equations than an evolutionary alternative, but required a fraction of computational resources. However, all symbolic regression methods substantially overfitted the training dataset. Finally, some issues remain open about how to best take advantage of the Bayesian framework of the model.

Keywords: Bayes, symbolic regression, machine learning

Research Internship Report
Bayesian Symbolic Regression

Introduction

Symbolic regression is a supervised machine learning technique that attempts to find mathematical expressions that describe relations between variables of a dataset. The mathematical expressions can be arbitrarily complex and are constructed from a prespecified set of features and operators (e.g. addition, multiplication, trigonometric functions, etc.). Machine learning methods are powerful for predictive purposes, but they are typically black-box models that are not interpretable (e.g. deep neural networks). However, researchers often desire interpretable models that explain something about the system that is being modeled, instead of just predicting the outcome. The main advantage of symbolic regression is that the resulting models are explicit and interpretable, since the output is just a mathematical expression.

Moreover, we can compose symbolic regression with any scientific model that is specified mathematically. For example, Dandekar et al. (2020) use symbolic regression and deep learning to extend an epidemiological model of COVID-19 spread (Lin et al., 2020). The model represents the spread of the pandemic as a system of differential equations, and captures our scientific knowledge about how contagious diseases spread. However, they did not have a good mathematical expression for a complex function that quantified the role of quarantine policies on the spread of the virus. With their approach they were able to recover—or discover—a mathematical expression for the role of quarantine policies with a plausible scientific interpretation. This example illustrates the possibilities that symbolic regression offers to bridge black-box machine learning techniques with scientific modeling of complex systems. In recent years there has been a movement of researchers arguing in favor of doing more formal modeling in psychological research (e.g. Guest and Martin, 2020; Robinaugh et al., 2021; van Rooij and Baggio, 2021). And, specially in psychological science where there is a lack established formal models, symbolic regression is a promising tool to aid in the development of formal models of psychological systems.

The space of possible mathematical expressions for any given problem is infinitely

large, and therefore it is not viable to explore it exhaustively. There are multiple methods that aim to recover mathematical expressions from a dataset, although they are not always called *symbolic regression* in the literature. Some of these methods use sparse regression (e.g. Brunton et al., 2016), neural networks (e.g. Sahoo et al., 2018) or a combination of both (e.g. Both et al., 2021). However, the most common approach to perform symbolic regression is to do a targeted search using evolutionary algorithms, which work by mimicking the evolution of a population of candidate expressions. Examples of this approach are the DEAP library (Fortin et al., 2012), the widely used proprietary software *Eureqa*[1] (Schmidt & Lipson, 2009) and the open-source implementation SymbolicRegression.jl/PySR (Cranmer, 2020). Evolutionary algorithms represent different mathematical expressions as symbolic trees and generate alternative expressions by mutating previous expressions. For example, a possible mutation of the symbolic tree depicted in Figure 1 could be to replace the *sin* node with another operator, or replacing the terminal node x_1 with a new branch that grows from a new operator. Every expression is assigned a fitness value, and the expressions with higher fitness from the population are selected to reproduce so that the population can evolve over multiple generations.

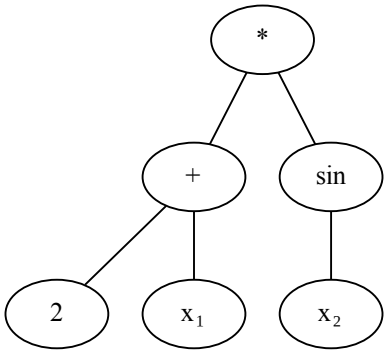


Figure 1

Symbolic tree that represents the expression $(2 + x_1) \cdot \sin x_2$.

A novel approach to perform symbolic regression is the use of Bayesian methods (Guimerà et al., 2020; Jin et al., 2020). These models use Markov chain Monte Carlo

[1] https://www.datarobot.com/nutonian/

(MCMC) samplers to explore the space of possible mathematical expressions and, similarly to the evolutionary algorithms, they represent the mathematical expressions as symbolic trees. In this case, instead of *mutating* the trees we say that MCMC samplers *jump* from one tree to another, although the modifications can be equivalent (e.g. replacing an operator or growing a new branch from where a terminal node used to be). However, the calculation of the probability of accepting a jump needs to satisfy a detailed balance that guarantees that the MCMC sampler will asymptotically converge to the posterior distribution of all possible mathematical expressions. Since these methods are in early stages of development, it is not clear yet how they compare to other symbolic regression models.

The current state of the art in expression discovery are models that combine symbolic regression and deep neural networks to encode prior information about the structure of the target mathematical expressions. Cranmer et al. (2020) fit neural networks that encode prior scientific knowledge about the structure of the target system, and then they recover mathematical expressions with evolutionary algorithms from the networks. With a different approach, Udrescu and Tegmark (2020) use a neural network to discover hidden structure that is common in physical formulas (e.g. coherent units, symmetry). These methods require less data, generalize better to out-of-sample data and have better predictive performance than just the neural networks or evolutionary algorithms on their own. Bayesian symbolic regression is equivalent in scope with the evolutionary or the sparse regression algorithms, and we could use either of them in combination with neural networks or directly over a dataset.

The goal of this internship was to implement an easy to use Bayesian symbolic regression program and to compare its performance against evolutionary methods. We build from the Bayesian symbolic regression model of Jin et al. (2020), which we describe in the next section alongside the modifications we have made. Our program, BayesianSR.jl[2], is available as an easy-to-use and performant package implemented in the Julia programming language (Bezanson et al., 2017).

[2] https://github.com/cobac/BayesianSR.jl

Bayesian symbolic regression algorithm

This section is an overview of the Bayesian symbolic regression model from Jin et al. (2020) and the two main modifications we have made to it. A complete mathematical specification of the model and other differences between their proposal and our implementation is included as an appendix. The model consists of a linear combination of mathematical expressions represented as symbolic trees:

$$y_i = \beta_0 + \beta_1 \Psi_1(\boldsymbol{x}_i) + \cdots + \beta_j \Psi_j(\boldsymbol{x}_i) + \cdots + \beta_K \Psi_K(\boldsymbol{x}_i) + \varepsilon_i \ .$$

Ψ_j is the j^{th} symbolic tree that represents a function of the features of the dataset. The total number of trees is defined by the hyperparameter K. $\boldsymbol{x}$ is the matrix of observations of features of the dataset and $\boldsymbol{y}$ is the outcome variable vector. The subscript i indexes particular observations of features and outcome variable. β_j are linear coefficients and ε_i is the residual for a particular observation. The residuals follow a normal distribution $N(0, \sigma^2)$. Every symbolic tree consists of nodes that can be of two types: either a terminal node representing a feature or an operator node representing an operation. There is a mandatory linear operator node that allows to include real-valued numbers in the symbolic trees.

For every iteration the MCMC sampler generates a new symbolic tree from the previous tree, choosing from a set of possible movements between trees. It also updates the real-valued numbers of the linear operator nodes. The linear coefficients β of each proposed sample are optimized via ordinary least squares. Each proposal can be accepted and the sampler *jumps* to the new sample, or rejected and the sampler stays in their current position. At every iteration the sampler only attempts to update one symbolic tree.

We have devised two modifications of the original algorithm. First, there are multiple symbolic trees that represent equivalent expressions (e.g. $x_1 + x_2 - x_1 + x_3$ and $x_2 + x_3$). To reduce the total number of possible mathematical expressions we can simplify every new expression that the MCMC sampler accepts. We have used the Julia package SymbolicUtils.jl (Gowda et al., 2021) to symbolically simplify every accepted sample. Secondly, instead of running only one MCMC chain we can run multiple chains initialized with different random expressions. The MCMC sampler has the option to generate a new proposal from a different chain at each iteration, instead of using the last sample of the current chain. This would allow

the program to explore multiple regions of posterior space at once and to use multiple CPU cores. We expected both modifications to make the MCMC sampler explore the posterior space of all possible expressions more efficiently.

Procedure

First, we compared the performance of the original program against our implementation. And secondly, we tested the modifications we propose (i.e. symbolic simplification step, multi-chain sampling and both combined) against the original algorithm and an evolutionary algorithm. The evolutionary algorithm we have used is SymbolicRegression.jl (Cranmer, 2020), a fast, parallel, distributed and open-source implementation.

To evaluate performance we measured two elements: accuracy of the models and computational cost. We used the Root Mean Squared Error (RMSE) to measure accuracy. To measure computational cost we have to take into account two factors. In first place, some algorithms can run on multiple CPU cores to speed up computation, while others cannot. Therefore, we used real time to measure computational cost instead of CPU time because we wanted to compare how efficient each algorithm is for a normal use case. If we had used CPU time we would had measured all parallel algorithms to perform multiple times worse than non-parallel ones. Secondly, we used a modified MCMC sampler that offloads chains from memory to run the simulations, which is unnecessary during a normal use case and creates significant overhead. To circumvent this issue we used a standardized time unit defined as the average time that a single iteration of the original Bayesian algorithm takes during normal usage. We calculated the average time that a single iteration takes for the rest of the algorithms and calculated proportionally the time they took in our custom units of time.

We tested all models with data generated from a standard set of functions (Expression 1) that have been used to benchmark other symbolic regression algorithms (Chen et al., 2015; Chen et al., 2016; Jin et al., 2020; Topchy, Punch, et al., 2001). We used the same training and test conditions as originally reported by Jin et al. (2020): data generated without noise from $U(-3, 3)$ for the training set, and from $U(-3, 3)$, $U(-6, 6)$ and $U(3, 6)$ for three different test sets. For each Bayesian algorithm we ran 50 simulations with datasets of 30 observations for

100,000 MCMC iterations for each function. Each model consisted of a linear combination of $K = 2$ symbolic trees. We provide a detailed description of all hyperparameters in the model specification appendix. We ran the evolutionary algorithm with 5 populations of 1000 members each for 3000 generations, for every function. For the rest of the options we used their default values. We allocated 5 CPU cores for all algorithms that supported parallel computation.

$$
\begin{aligned}
f_1(x_0, x_1) &= 2.5x_0^4 - 1.3x_0^3 + 0.5x_1^2 - 1.7x_1 \\
f_2(x_0, x_1) &= 8x_0^2 + 8x_1^3 - 15 \\
f_3(x_0, x_1) &= 0.2x_0^3 + 0.5x_1^3 - 1.2x_1 - 0.5x_0 \\
f_4(x_0, x_1) &= 1.5\exp(x_0) + 5\cos(x_1) \\
f_5(x_0, x_1) &= 6.0\sin(x_0)\cos(x_1) \\
f_6(x_0, x_1) &= 1.35x_0x_1 + 5.5\sin\left[(x_0 - 1)(x_1 - 1)\right]
\end{aligned} \tag{1}
$$

Results

First, we tested our implementation against the algorithm from Jin et al. (2020) written in Python and we observed an speedup of x2188. Our model in average generates one sample each $336\mu s$, while theirs took $735ms$.

The first modification to the original algorithm we wanted to test was to simplify symbolically each accepted sample. This extra step has a computational overhead, but we thought it might improve the performance of the program. Our rationale was that by simplifying all expressions we would be effectively reducing the space of possible expressions that the sampler had to explore. However, in Figure 2 we see that this step occasionally causes the sampler to get stuck on bad expressions, as evidenced by the stair-like pattern seen for equations nos. 1, 4 and 5. We also see for all equations sudden peaks. This occurs because of differences in how floating point errors are propagated between the generated expression and the simplified expression. When a floatint point number exceeds their maximum size in bytes, the Julia language represents it with an infinity object. Some expressions that were computable become uncomputable after the simplification step (e.g. if a component becomes

sin(Inf)). Only for Equations nos. 2 and 6 the modification seems to consistently outperform the original algorithm, and we conclude that in general the simplification step in its current form is not worth the additional computational overhead.

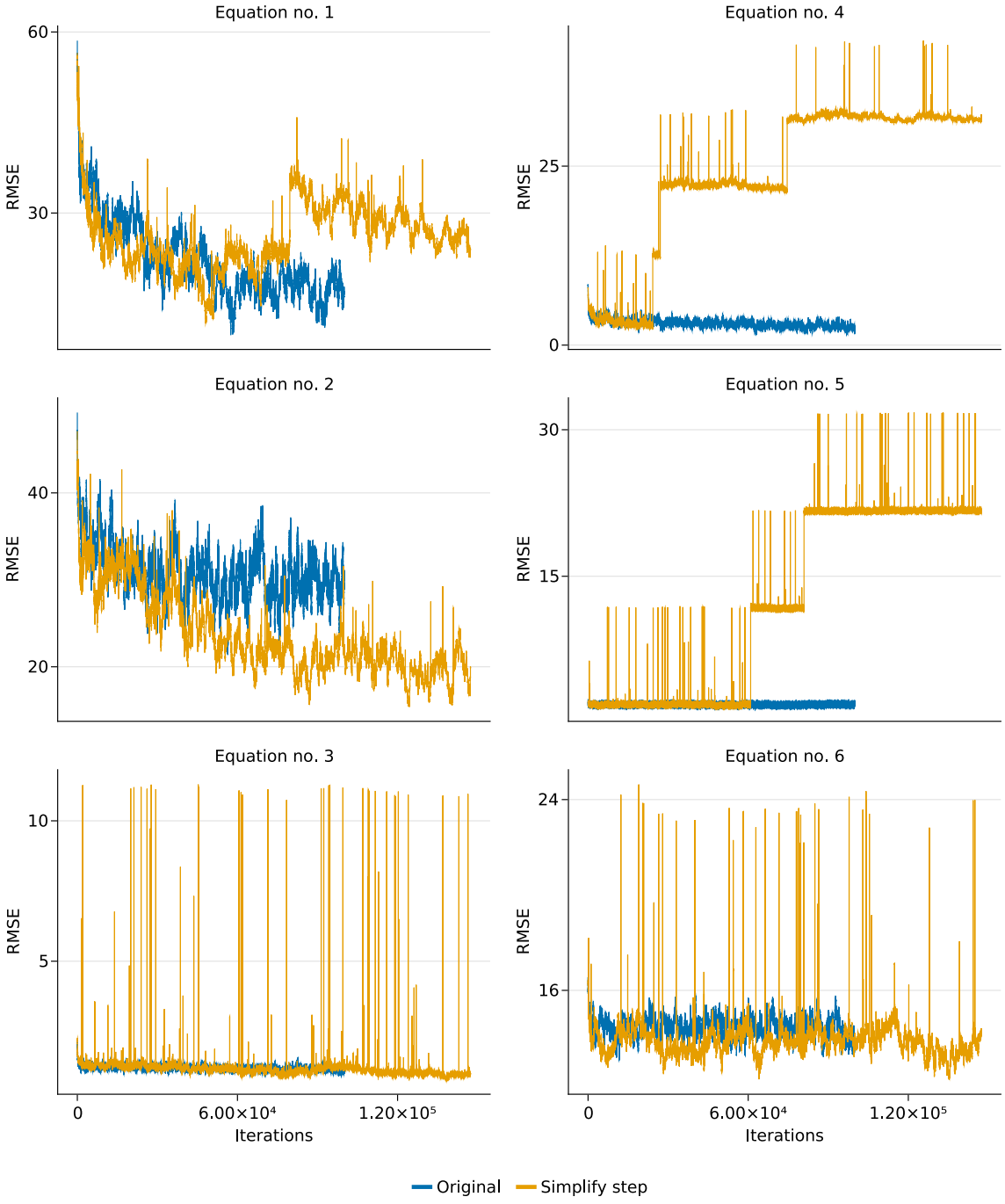


Figure 2

Average RMSE across all simulations through runtime. We show the original algorithm and the modified algorithm with a symbolic simplification step. We capped RMSE values at 500 before averaging so that infinities could be displayed.

The second modification we wanted to test was to use multiple MCMC chains, and to allow with some probability each chain to generate a sample from a different chain. This extra step allows the algorithm to run on parallel and does not have noticeable overhead on machines with multiple CPU cores. We expected that running multiple chains would increase the performance of the program, since it would be able to explore multiple areas of posterior space simultaneously. In Figure 3 we observe that the modification clearly outperforms the original algorithm for all equations. In Figure 4 we see an example of how the chains from the multi-chain algorithm quickly find and remain exploring more optimal areas of posterior space. Additionally, we see in Figure 5 that the algorithm with multi-chain sampling and symbolic simplification has the same issues that the algorithm with only the simplification step: we observe the stair-like anomalies and the sudden peaks. We conclude that multi-chain sampling outperforms the original algorithm and that, even in environments without access to CPUs with multiple cores, this modification is the preferred algorithm amongst all versions.

However, our Bayesian symbolic regression algorithm is substantially overfitting the data from the training set. In Figure 6 we see that our algorithm cannot even model properly the testing set generated from the same distribution as the training data. The distributions of RMSE remain mostly constant or random across training, maybe with an exception for the third equation. It is noteworthy that in some cases the initial random guesses of the algorithm are in average better models of the testing sets than samples further on the MCMC chains. If we compare the results from the multi-chain algorithm with the original algorithm (Figure 7) we observe that each of them perform slightly better than the other in some cases, but in both cases the programs significantly overfit the training set. These results are contradictory with what Jin et al. (2020) report. In their article the expressions that the Bayesian symbolic regression program discovers—under the same conditions—model equally well the training data and the testing data generated from the same distribution.

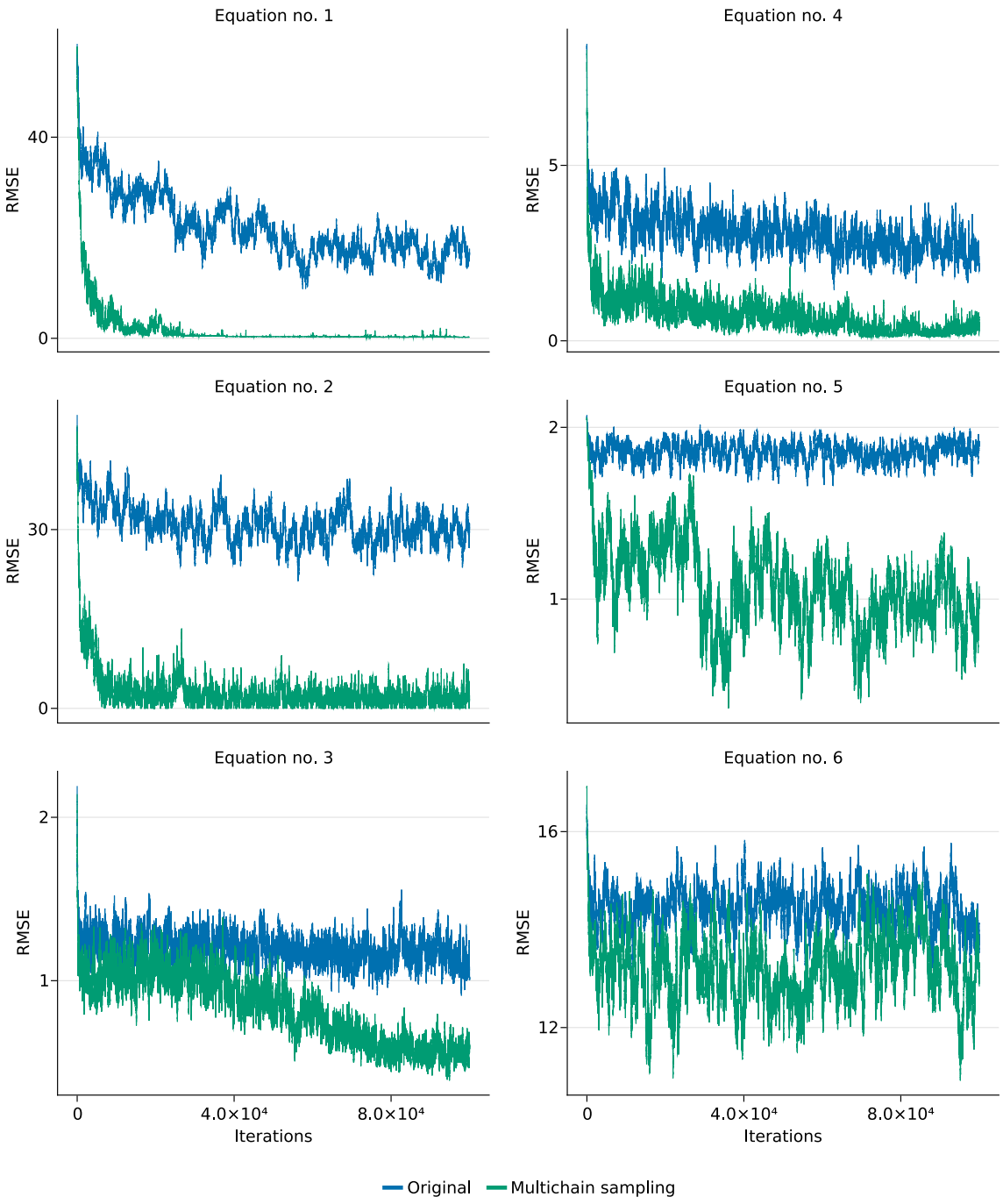
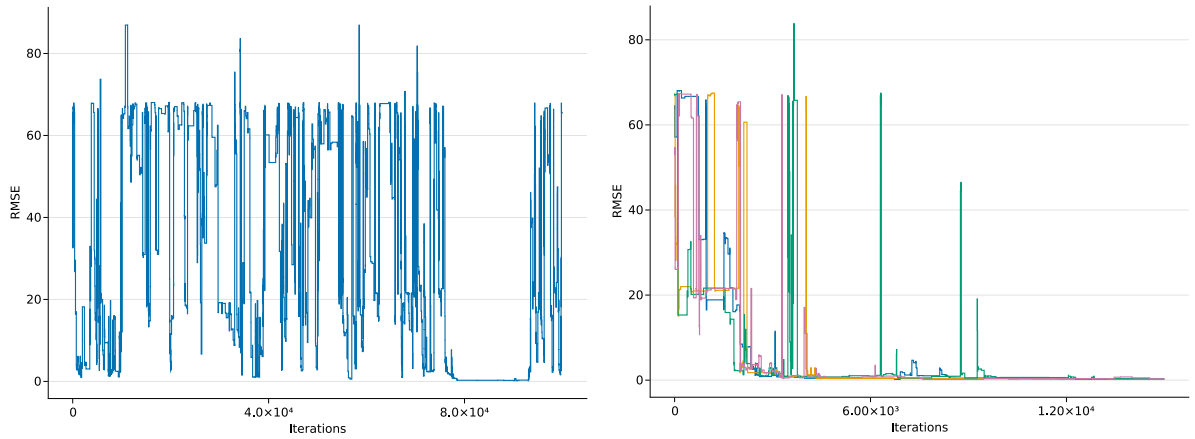


Figure 3

Average RMSE across all simulations through runtime. We show the original algorithm and the modified algorithm with multi-chain sampling. We capped RMSE values at 500 before averaging so that infinities could be displayed.



(a) *The MCMC chain from the original implementation.*

(b) *The five MCMC chains from the multi-chain sampler.*

Figure 4

RMSE of the samples from different algorithms. Both are from the first simulation with data generated from the first equation.

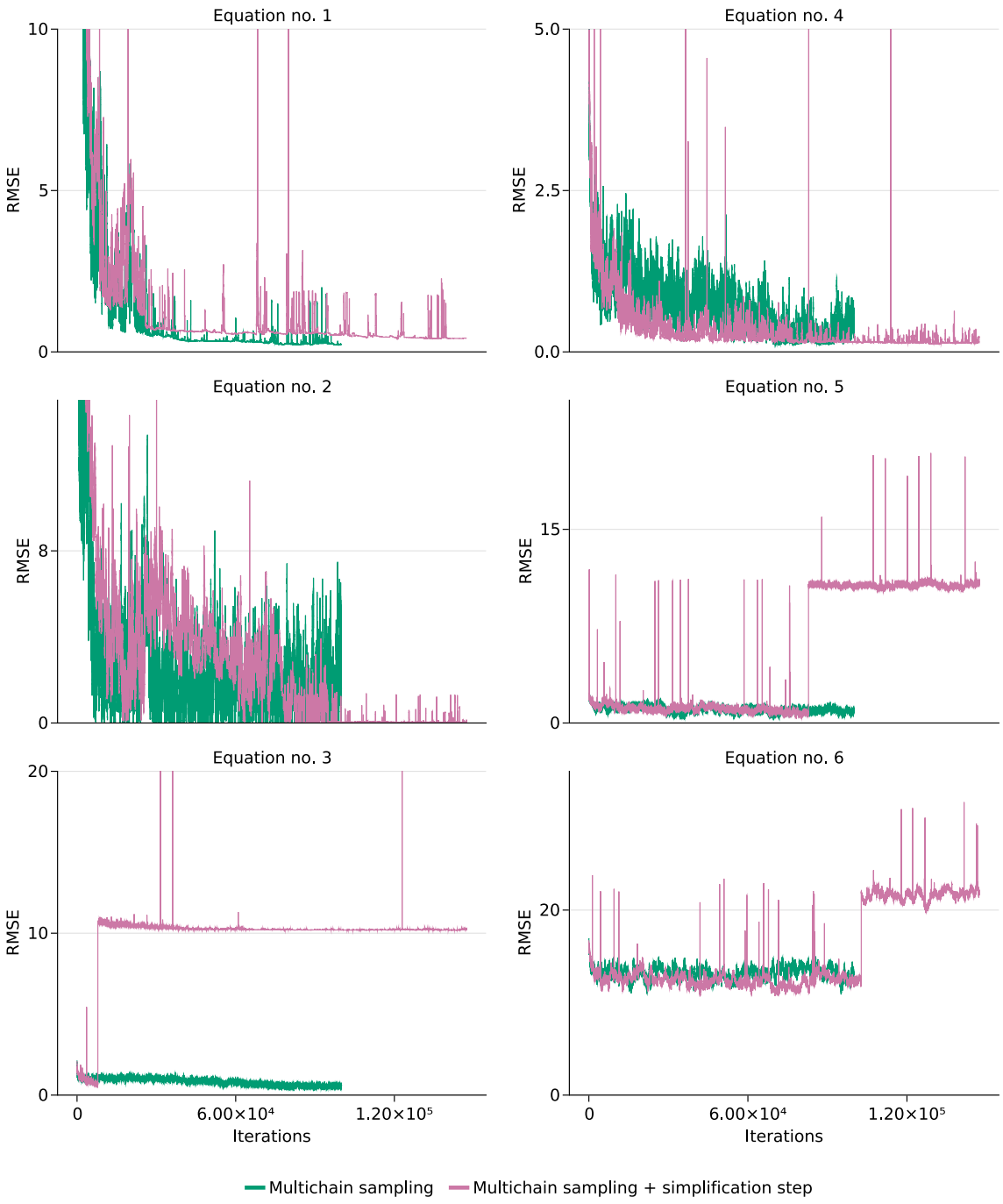


Figure 5

Average RMSE across all simulations through runtime. We show the modified algorithms with multi-chain sampling alone and combined with symbolic simplification. We capped RMSE values at 500 before averaging so that infinities could be displayed. The RMSE axis is also zoomed-in to show greater detail.

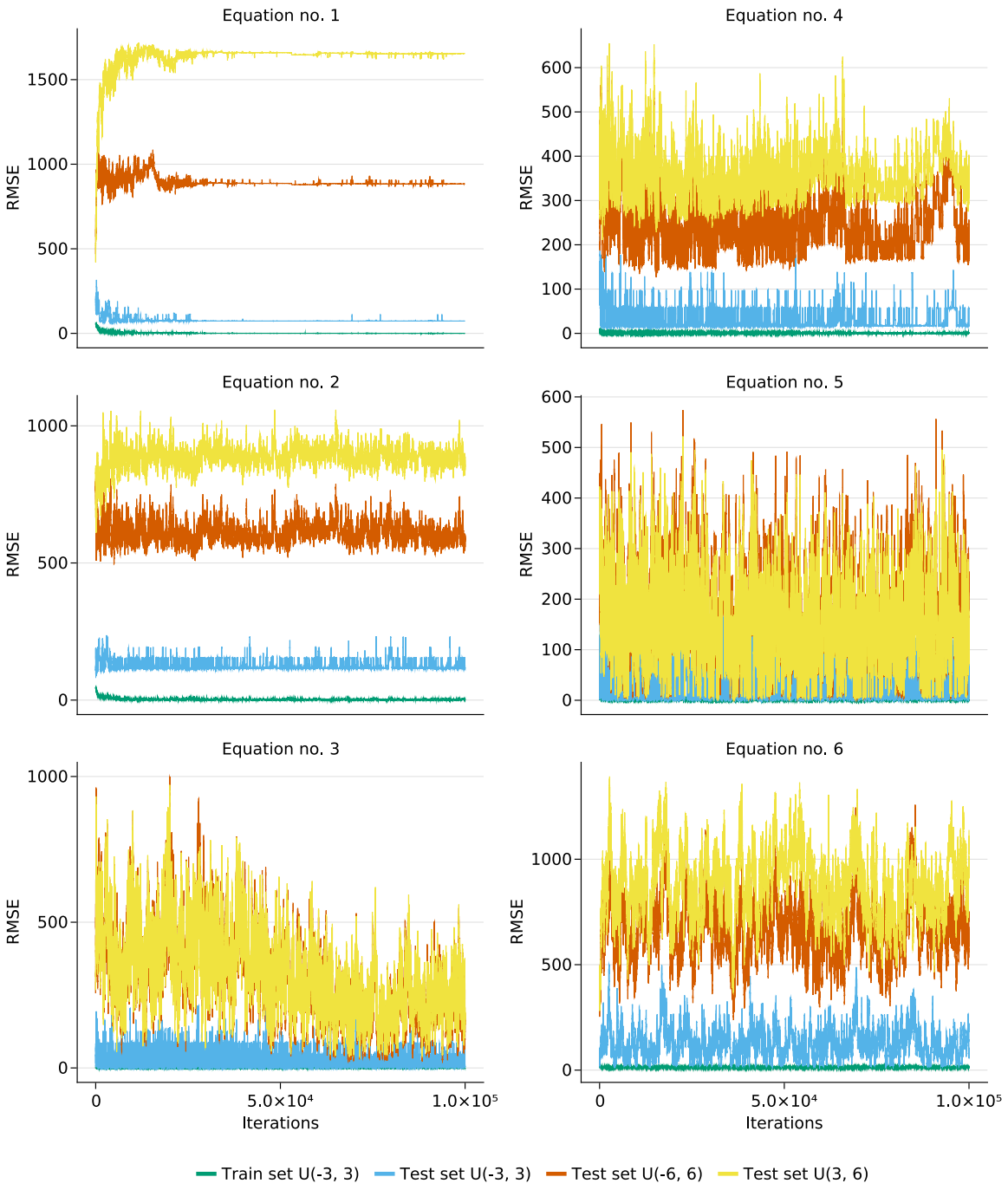


Figure 6

Average RMSE across all simulations through runtime of the multi-chain algorithm. We show values generated from the training and test sets. We capped RMSE values at 2000 before averaging so that infinities could be displayed.

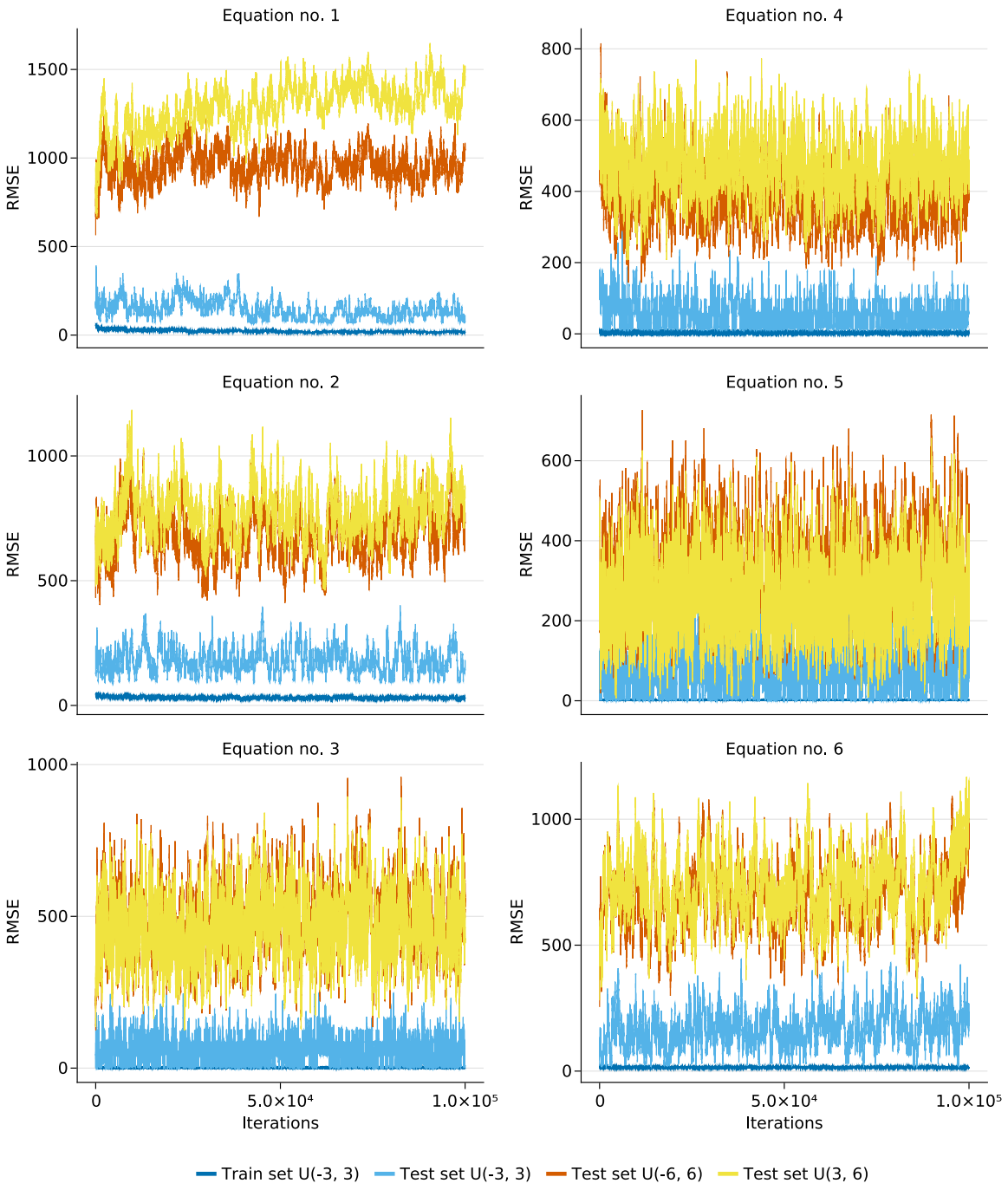


Figure 7

Average RMSE across all simulations through runtime of the original algorithm. We show values generated from the training and test sets. We capped RMSE values at 2000 before averaging so that infinities could be displayed.

Finally, we compare the performance of the evolutionary and the Bayesian algorithms. However, the output of the evolutionary algorithm is different from the output of the Bayesian algorithm. Instead of a MCMC chain it reports the Pareto frontier between the RMSE and the complexity of the expressions. In Figure 8 we see that the evolutionary algorithm is able to discover equations that model the data from the training set with similar errors as the Bayesian algorithm. Nevertheless, it is also overfitting, since the RMSE values for all testing sets are greater than the values corresponding to the lowest complexity. Those simplest expressions are essentially semi-random guesses, since they are symbolic trees with a single node representing a real number. Moreover, this algorithm is designed to run on a different scale than the Bayesian programs, both in terms of simultaneous parallel processes and time. We ran the algorithm for the minimum amount of time—while keeping the number of CPU cores consistent—that produced results similar to those of the Bayesian algorithm. Whereas a complete run of the Bayesian algorithm took 33.6s in average the evolutionary algorithm took 364s, a whole order of magnitude greater. These differences make it hard to compare both models, but we have shown that under the conditions we have tested, the Bayesian symbolic regression model can discover expressions as good as the evolutionary algorithm in a fraction of the time.

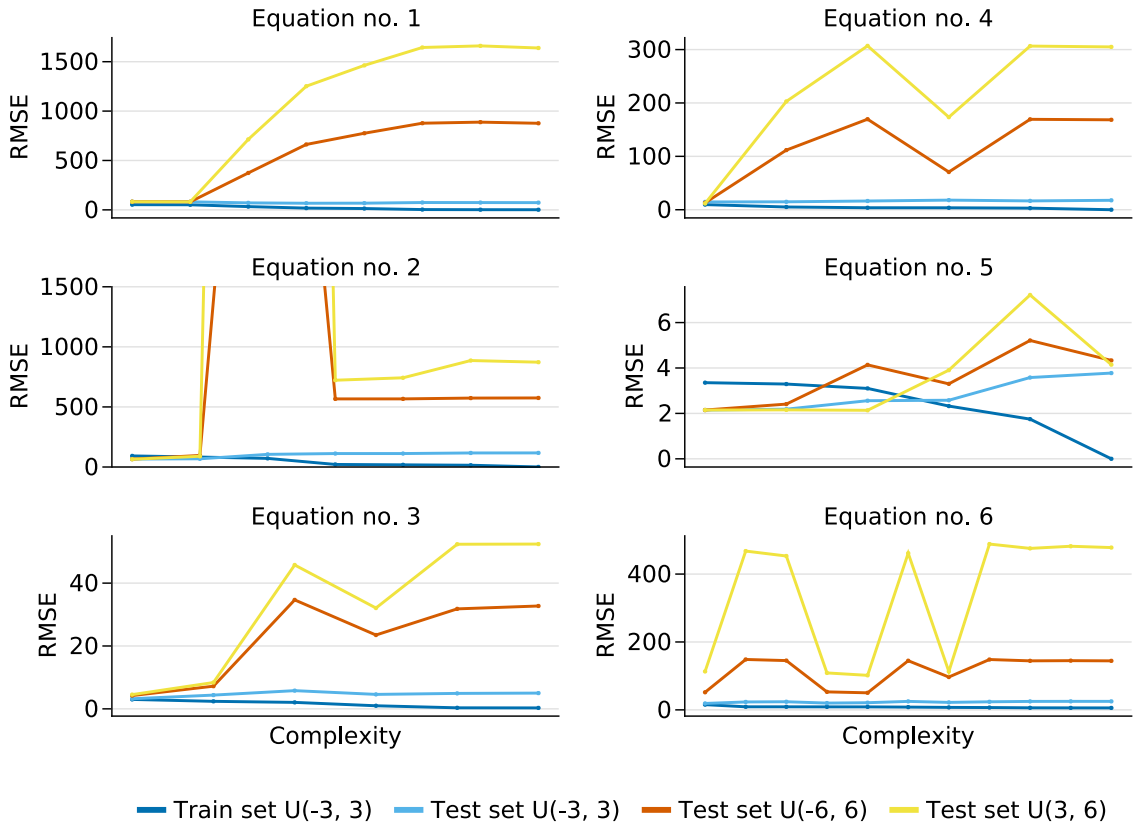


Figure 8

Pareto frontiers of RMSE and complexity of the expressions reported by the evolutionary algorithm for each equation. Complexity is an ordinal varible measured in number of nodes of the symbolic trees. Exact values are ommited to avoid gaps and make the visualization clearer. For equation no. 2 the RMSE axis is zoomed-in and the missing data points are in the 10^4 *order of magnitude.*

Discussion

Although we have shown that the Bayesian symbolic regression algorithm finds equally good expressions as the evolutionary program on a fraction of the time, our conclusions are limited to the conditions we have tested. Since the evolutionary algorithm is conceived to run on a larger timescale, it would be appropriate to compare both methods in the future under the recommended conditions of the evolutionary algorithm. Moreover, the issue of overfitting might had been alleviated if we had used bigger sample sizes for each simulation. In this case we prioritized running our simulations under the same conditions as reported by Jin et al. (2020), in order to compare both implementations of the same model. However, another main limitation is that it is not clear which hyperpameters they used during their simulations. They do not report them in their article and it is not clear from their source code either. We expect hyperparameters to have an important effect on the outcome of the model. They report that the total number of symbolic trees K seems to not affect the results, since the algorithm is able to adapt the complexity of each tree to compensate for different values of K. But the hyperparameters that govern how new symbolic trees grow determine how the complexity of the expressions is distributed. Maybe if we had used hyperparameters that favored less complex expressions we would have observed less overfitting, matching what Jin et al. (2020) report.

The main topic we have not addressed in this report and remains pending for the future is how to select output expressions from the MCMC chains. What Jin et al. (2020) do is report the last expression from the chains after a prespecified number of iterations. However, we have observed (e.g. in Figure 4 a) that the last expression does not have to be a particularly good one. We could choose instead the equation with highest fit from a chain, but this approach would be biased towards more complex expressions. In this case, if the sampler had discovered the correct expression it is guaranteed to have the best fit since the datasets were generated without noise. But during real uses cases, or simulations with noisy data, this approach would worsen the overfitting issues. Alternatively, to restrain overfitting we could report the Pareto frontier between fit and complexity like SymbolicRegression.jl (Cranmer, 2020) do. Nevertheless, ideally we would use an approach that allowed us to take advantage of the Bayesian components of the model and generate confidence distributions. For

predictive purposes this is easy since we can just generate predictions from all the samples of a chain—possibly after a warm-up interval. But it is not clear how to generate confidence distributions if our objective is to discover interpretable expressions. If there is a set of tree structures that the MCMC chains explore frequently, we can use their relative frequencies as confidence distributions. However, we would have to take into consideration the distribution of their linear operator node coefficients. If we decided to commit to this approach it would be beneficial to separate the MCMC sampler into two alternating movements: sampling a new tree structure and then sampling a new set of linear operator coefficients.

We are aware that our analysis of the performance of the methods is simplistic, since we have focused mainly on predictive accuracy. In the future, after developing more mature symbolic regression MCMC samplers, we would have to examine how interpretable are the expressions that the algorithm can discover considering trade-offs between fit, computational resources and interpretability. After all, from our perspective the major advantage of symbolic regression over other methods is its ability to aid in discovering mathematical expressions with plausible scientific interpretations. We conclude from our preliminary results that Bayesian symbolic regression is a promising method that seems to be able to match the performance of the current most common alternatives. Hopefully it will become in the near future a useful tool for researchers that wish to compose machine learning techniques with their scientific models.

Materials

All project materials are available in the following Open Science Framework repository: https://osf.io/p8bg5/ . It includes code for all simulations and analyses, simulations raw output and intermediary analysis results.

References

Bezanson, J., Edelman, A., Karpinski, S., & Shah, V. B. (2017). Julia: A fresh approach to numerical computing. *SIAM Review*, *59*(1), 65–98. https://doi.org/10.1137/141000671

Both, G.-J., Choudhury, S., Sens, P., & Kusters, R. (2021). Deepmod: Deep learning for model discovery in noisy data. *Journal of Computational Physics*, *428*, 109985. https://doi.org/10.1016/j.jcp.2020.109985

Brunton, S. L., Proctor, J. L., & Kutz, J. N. (2016). Discovering governing equations from data by sparse identification of nonlinear dynamical systems. *Proceedings of the national academy of sciences*, *113*(15), 3932–3937. https://doi.org/10.1073/pnas.1517384113

Chen, Q., Xue, B., Shang, L., & Zhang, M. (2016). Improving generalisation of genetic programming for symbolic regression with structural risk minimisation. *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, 709–716. https://doi.org/10.1145/2908812.2908842

Chen, Q., Xue, B., & Zhang, M. (2015). Generalisation and domain adaptation in gp with gradient descent for symbolic regression. *2015 IEEE congress on evolutionary computation (CEC)*, 1137–1144. https://doi.org/10.1109/CEC.2015.7257017

Cranmer, M. (2020). *Pysr: Fast & parallelized symbolic regression in python/julia*. Zenodo. https://doi.org/10.5281/zenodo.4041459

Cranmer, M., Sanchez-Gonzalez, A., Battaglia, P., Xu, R., Cranmer, K., Spergel, D., & Ho, S. (2020). Discovering symbolic models from deep learning with inductive biases. *CoRR*. http://arxiv.org/abs/2006.11287v2

Dandekar, R., Rackauckas, C., & Barbastathis, G. (2020). A machine learning-aided global diagnostic and comparative tool to assess effect of quarantine control in COVID-19 spread. *Patterns*, *1*(9), 100145. https://doi.org/10.1016/j.patter.2020.100145

Fortin, F.-A., De Rainville, F.-M., Gardner, M.-A. G., Parizeau, M., & Gagné, C. (2012). Deap: Evolutionary algorithms made easy. *The Journal of Machine Learning Research*, *13*(1), 2171–2175. http://jmlr.org/papers/v13/fortin12a.html

Gowda, S., Ma, Y., Cheli, A., Gwozdz, M., Shah, V. B., Edelman, A., & Rackauckas, C. (2021). High-performance symbolic-numerics via multiple dispatch. http://arxiv.org/abs/2105.03949v2

Green, P. J. (1995). Reversible jump markov chain monte carlo computation and bayesian model determination. *Biometrika*, *82*(4), 711–732. https://doi.org/10.1093/biomet/82.4.711

Guest, O., & Martin, A. E. (2020). How computational modeling can force theory building in psychological science. https://doi.org/10.31234/osf.io/rybh9

Guimerà, R., Reichardt, I., Aguilar-Mogas, A., Massucci, F. A., Miranda, M., Pallarès, J., & Sales-Pardo, M. (2020). A Bayesian machine scientist to aid in the solution of challenging scientific problems. *Science Advances*, *6*(5), eaav6971. https://doi.org/10.1126/sciadv.aav6971

Jin, Y., Fu, W., Kang, J., Guo, J., & Guo, J. (2020). Bayesian symbolic regression. http://arxiv.org/abs/1910.08892v3

Lin, Q., Zhao, S., Gao, D., Lou, Y., Yang, S., Musa, S. S., Wang, M. H., Cai, Y., Wang, W., Yang, L., et al. (2020). A conceptual model for the coronavirus disease 2019 (COVID-19) outbreak in wuhan, china with individual reaction and governmental action. *International journal of infectious diseases*, *93*, 211–216. https://doi.org/10.1016/j.ijid.2020.02.058

Robinaugh, D. J., Haslbeck, J. M. B., Ryan, O., Fried, E. I., & Waldorp, L. J. (2021). Invisible hands and fine calipers: A call to use formal theory as a toolkit for theory construction. *Perspectives on Psychological Science*, *16*(4), 725–743. https://doi.org/10.1177/1745691620974697

Sahoo, S., Lampert, C., & Martius, G. (2018). Learning equations for extrapolation and control. *International Conference on Machine Learning*, 4442–4450. http://proceedings.mlr.press/v80/sahoo18a.html

Schmidt, M., & Lipson, H. (2009). Distilling free-form natural laws from experimental data. *Science*, *324*(5923), 81–85. https://doi.org/10.1126/science.1165893

Topchy, A., Punch, W. F. et al. (2001). Faster genetic programming based on local gradient search of numeric leaf values. *Proceedings of the 3rd Annual Conference on Genetic and Evolutionary Computation*, 155–162. https://doi.org/10.5555/2955239.2955258

Udrescu, S.-M., & Tegmark, M. (2020). AI Feynman : A physics-inspired method for symbolic regression. *Science Advances*, *6*(16). https://doi.org/10.1126/sciadv.aay2631

van Rooij, I., & Baggio, G. (2021). Theory before the test: How to build high-verisimilitude explanatory theories in psychological science. *Perspectives on Psychological Science*, *16*(4), 682–697. https://doi.org/10.1177/1745691620970604

Appendix

Bayesian symbolic regression model specification

This section is a complete mathematical specification of the Bayesian symbolic regression model from Jin et al. (2020) and the differences between the original program[3] and our implementation[4] .

General model structure

The model consists of a linear combination of mathematical expressions represented as symbolic trees:

$$y_i = \beta_0 + \beta_1 \Psi_1(\boldsymbol{x}_i) + \cdots + \beta_j \Psi_j(\boldsymbol{x}_i) + \cdots + \beta_K \Psi_K(\boldsymbol{x}_i) + \varepsilon_i \ .$$

Ψ_j is the j^{th} symbolic tree that represents a function of the features of the dataset. The total number of trees is defined by the hyperparameter K. $\boldsymbol{x}$ is the matrix of observations of features of the dataset and $\boldsymbol{y}$ is the outcome variable vector. The subscript i indexes particular observations of features and outcome variable. β_j are linear coefficients and ε_i is the residual for a particular observation. The residuals follow a normal distribution $N(0, \sigma^2)$.

Every symbolic tree consists of nodes that can be of two types: either a terminal node representing a feature or an operator node representing an operation. Operator nodes can be either binary (e.g. addition, multiplication) or unary (e.g. sin, x^2), and any combination of operators is possible. There is a mandatory linear operator node $lt(x) = a + bx$ that allows to include real-valued numbers in the symbolic trees. Each tree Ψ is characterized by its structure S and by the set of linear operator coefficients of all linear operator nodes Θ.

Prior distributions of the model

We can assign prior probabilities for all operators and features. By default we select from the sets of operators and features uniformly, but it is possible to assign weights to each set.

[3] https://github.com/ying531/MCMC-SymReg

[4] https://github.com/cobac/BayesianSR.jl

We also specify a tree structure prior distribution $p(S)$ that governs how to grow a new symbolic tree. To insert a new node η we select either a new operator or terminal node. The probability of selecting a new operator depends on its depth d_η (for the root node of a tree $d = 1$) and two hyperparameters α and β:

$$f(\eta, S) = \alpha\left(1 + d_\eta\right)^{-\beta}.$$

We ran our simulations with $\alpha = 2$ and $\beta = 1$. $p(S)$ is the joint distribution of all nodes of a symbolic tree.

Linear coefficients a and b of the linear operator nodes have prior distributions

$$a_\eta \sim N(0, \sigma_a^2) \text{ and } b_\eta \sim N(1, \sigma_b^2)$$

respectively. The prior distribution of the linear coefficients of a symbolic tree $p(\Theta|S)$ is the joint distribution of all the linear operator coefficients. The variances of the linear coefficients $\sigma_\Theta^2 = \left(\sigma_a^2, \sigma_b^2\right)$ have prior distributions

$$\sigma_a^2 \sim IG\left(\nu_a/2, \nu_a\lambda_a/2\right) \text{ and } \sigma_b^2 \sim IG\left(\nu_b/2, \nu_b\lambda_b/2\right),$$

where IG denotes the inverse-gamma distribution and ν_a, λ_a, ν_b and λ_b are prespecified hyperparameters. We ran our simulations with all of them being 1.

The prior distribution of the variance of the residuals $p(\sigma^2)$ is $IG(\nu/2, \nu\lambda/2)$, where ν and λ are prespecified hyperparameters. We ran our simulations with both of them being 1.

Posterior sampling

For every iteration of the program the MCMC sampler generates a proposal and either accepts or rejects the new sample. Variables with superscript $X^{(t)}$ denote that they belong to the last sample of the MCMC chain and variables with superscript X^* denote that they are from the proposal. At every iteration the sampler only attempts to update one symbolic tree, and it iterates through all symbolic trees in order. In the algorithms that use multi-chain sampling we can run multiple chains in parallel, and at every iteration there is a probability that the MCMC sampler will generate a proposal from a different chain selected at random. We always ran 5 chains in our simulations with a 0.05 probability of inter-chain sampling.

Possible movements between trees

First, the MCMC sampler proposes a new tree structure S^* by modifying the previous symbolic tree $S^{(t)}$. There are 6 possible movements between trees, and at every iteration the MCMC samplers only perform a single movement.

Stay There is a $p_0 = \frac{N_l}{4(N_l+3)}$ probability that the structure of the tree remains unchanged, where N_l is the number of linear operator nodes of the tree.

Grow There is a $p_g = \frac{1-p_0}{3} \cdot \min\left\{1, \frac{8}{N_o+2}\right\}$ probability that the structure of the tree grows, where N_o is the number of operator nodes in the tree. To grow a tree we select a random terminal node and grow a new branch from the tree structure prior distribution.

Prune There is a $p_p = \frac{1-p_0}{3} - p_g$ probability that the tree structure is pruned. To prune a tree we select a random operator node and substitute it for a terminal node selected from the terminal nodes prior distribution.

Delete There is a $p_d = \frac{1-p_0}{3} \cdot \frac{N_c}{N_c+3}$ probability that a node is deleted, where N_c is the number of nodes that are eligible candidates for deletion. Nodes can be deleted if they are operator nodes, and if they are the root node they need to have at least one operator node child. If the selected node is unary we replace it for its child. If its binary and non-root we replace it for one of its children at random. If its binary and root, we replace it for one of its operator children at random.

Insert There is a $p_i = \frac{1-p_0}{3} - p_d$ probability of inserting a new operator node drawn from the operator nodes prior distribution between a random node and its parent. If we insert a binary operator node, the original node becomes its first child and a new child is generated from the tree structure prior distribution.

Reassign operator There is a $p_{ro} = \frac{1-p_0}{6}$ probability of reassigning a random operator node with a new operator drawn from the operator nodes prior distribution. If we transition from an unary operator to a binary operator, the old child becomes its first child and a new child is generated from the tree structure prior distribution. If we transition from a binary operator to a unary operator, we keep the first child.

Reassign feature There is a $p_{rf} = \frac{1-p_0}{6}$ probability of reassigning a random terminal node with a new terminal node drawn from the terminal nodes prior distribution.

After each movement we can calculate $q(S^*|S^{(t)})$, the probability of moving from the old tree structure $S^{(t)}$ to the new tree structure S^*, as well as the probability of the movement in reverse $q(S^{(t)}|S^*)$. *Grow and prune* and *delete and insert* movement pairs are complementary, since they allow to reverse the other movement of the pair. To calculate $q(\cdot)$ we need to take into account the probability of selecting each movement, as well as the probabilities of each of the intermediary steps of each movement (e.g. choosing a specific node from the tree structure or prior distributions).

Updating the linear operator coefficients

After generating a new tree structure the MCMC sampler generates a new set of linear operator node coefficients Θ^*. However, the new tree structure may contain a different number of linear operator nodes than the old symbolic tree. If this is the case, Θ will have different dimensions between trees, and we need to use reversible jump Markov chain Monte Carlo (RJMCMC, Green, 1995) to update the linear coefficients. Three different situations can arise:

No change If the number of linear operator nodes is the same in S^* and in $S^{(t)}$ no RJMCMC is required. In the original algorithm the set of new coefficients Θ^* was generated from the prior distributions of the coefficients. However, we have chosen to instead generate proposals centered around the previous values of the coefficients adding a random value drawn from a $N(0, 1)$ distribution.

Expansion If the number of linear operator nodes is greater in S^* than in $S^{(t)}$ we need to use RJMCMC. We sample auxiliary variables $U = (u_\Theta, u_n)$, where $\dim(u_\Theta) = \dim(\Theta^{(t)})$ and $\dim(u_n) = \dim(\Theta^*) - \dim(\Theta^{(t)})$. We sample σ_a^2 and σ_b^2 from the variances prior distributions, and u_Θ and u_n are independently sampled from $N(1, \sigma_a^2)$ and $N(0, \sigma_b^2)$ respectively. Next, we define a function $j_e(\Theta^{(t)}, U)$ that generates the new set of parameters Θ^* and the auxiliary variables U^*:

$$\Theta^* = \left(\frac{\Theta^{(t)} + u_\Theta}{2}, u_n\right), \quad U^* = \frac{\Theta^{(t)} - u_\Theta}{2}.$$

Shrinkage If there are less linear operator nodes in S^* than in $S^{(t)}$ we also need to use RJMCMC. We divide the original linear coefficients set $\Theta^{(t)}$ into Θ_0 and Θ_d, where Θ_d are the coefficients of nodes that are dropped. We sample σ_a^2 and σ_b^2 from the variances prior distributions, and the auxiliary variables U from $N(0, \sigma_a^2)$ and $N(0, \sigma_b^2)$ respectively, with $\dim(U) = \dim(\Theta_0)$. Next, we define a function $j_s(\Theta^{(t)}, U)$ that generates the new set of parameters Θ^* and the auxiliary variables U^*:

$$\Theta^* = \Theta_0 + U, \;\; U^* = (\Theta_0 - U, \Theta_d)\,.$$

Note that $\dim(\Theta^{(t)}) + \dim(U) = \dim(\Theta^*) + \dim(U^*)$, which is necessary so that the Jacobian determinant in Expression 3 is computable. $h(\{U, U^*\}|\Theta^{(t)}, S^{(t)}S^*)$ denotes the probability distributions of the auxiliary variables U and U^*.

Accepting new samples

The last two steps that the MCMC performs are sampling a new variance $sigma^2)^*$ from the variance prior distribution and optimizing via ordinary least squares the linear coefficients β. After generating a new proposal the MCMC sampler has to decide whether to accept it, and add the new sample to the MCMC chain, or reject it, and add a new copy of the previous sample to the MCMC chain. The probability of accepting the proposed sample is $\min\{1, R\}$, where R is defined in Expression 2 for the case when no RJMCMC is required and in Expression 3 for the case with RJMCMC. This probability of accepting a new sample guarantees that the MCMC chains will asymptotically converge with the posterior distribution of all possible models. For brevity we denote all the sampled variances of the model as $\Sigma = (\sigma^2, \sigma_\Theta^2)$, and $p(\Sigma)$ their joint prior distribution. $f\left[y|\text{OLS}(\boldsymbol{x}, S, \Theta), \Sigma\right]$ is the value of the likelihood function of the model given a particular sample. In the versions of the algorithm that include a symbolic simplification step, newly accepted symbolic trees are simplified symbolically using the Julia package SymbolicUtils.jl (Gowda et al., 2021) before adding the sample to the MCMC chain.

$$R = \frac{f\left[y|\text{OLS}\left(\boldsymbol{x}, S^*, \Theta^*\right), \Sigma^*\right] p(S^*) q(S^{(t)}|S^*) p(\Theta^*|S^*) p(\Sigma^*)}{f\left[y|\text{OLS}\left(\boldsymbol{x}, S^{(t)}, \Theta^{(t)}\right), \Sigma^{(t)}\right] p(S^{(t)}) q(S^*|S^{(t)}) p(\Theta^{(t)}|S^{(t)}) p(\Sigma^{(t)})} \tag{2}$$

$$R = \frac{f\left[y|\text{OLS}\left(\boldsymbol{x}, S^*, \Theta^*\right), \Sigma^*\right] p(S^*) q(S^{(t)}|S^*) p(\Theta^*|S^*) p(\Sigma^*) h\left(U^*|\Theta^{(t)}, S^*, S^{(t)}\right)}{f\left[y|\text{OLS}\left(\boldsymbol{x}, S^{(t)}, \Theta^{(t)}\right), \Sigma^{(t)}\right] p(S^{(t)}) q(S^*|S^{(t)}) p(\Theta^{(t)}|S^{(t)}) p(\Sigma^{(t)}) h\left(U|\Theta^{(t)}, S^*, S^{(t)}\right)} \cdot \left| \frac{\partial j\left(\Theta^{(t)}, U|S^{(t)}, S^*\right)}{\partial\left(\Theta^{(t)}, U\right)} \right| \tag{3}$$

The Jacobian determinant from Expression 3 is equivalent to $2^{-\dim(\Theta^{(t)})}$ in the expansion case, and to $2^{\dim(\Theta^*)}$ in the shrinkage case.

Expressions 2 & 3 differ from their counterparts from Jin et al. (2020), which is a work in progress paper. In their version they omitted some of the components that we include to calculate the ratios. Moreover, they only add accepted samples to the MCMC chain, while we add a copy of the previous sample if the proposal is rejected. We believe our implementation is correct and it guarantees the detailed balance required for the MCMC sampler to asymptotically converge with the posterior distribution of all possible models. We have contacted the original authors[5] and they will consider our comments in future versions of their paper.

[5] https://github.com/ying531/MCMC-SymReg/issues/2