Reducing Overparameterization of Symbolic Regression Models with Equality Saturation

Fabrício Olivetti de França
Universidade Federal do ABC
Center for Mathematics, Computing and Cognition
Heuristics, Analysis and Learning Laboratory (HAL)
Santo André, SP, Brazil
folivetti@ufabc.edu.br

Gabriel Kronberger
Josef Ressel Center for Symbolic Regression, Heuristic and Evolutionary Algorithms Laboratory, University of Applied Sciences Upper Austria,
Hagenberg, Austria
gabriel.kronberger@fh-hagenberg.at

ABSTRACT

Overparameterized models in regression analysis are often harder to interpret and can be harder to fit because of ill-conditioning. Genetic programming is prone to overparameterized models as it evolves the structure of the model without taking the location of parameters into account. One way to alleviate this is rewriting the expression and merging the redundant fitting parameters. In this paper we propose the use of equality saturation to alleviate overparameterization. We first notice that all the tested GP implementations suffer from overparameterization to different extents and then show that equality saturation together with a small set of rewriting rules is capable of reducing the number of fitting parameters to a minimum with a high probability. Compared to one of the few available alternatives, Sympy, it produces much better and consistent results. These results lead to different possible future investigations such as the simplification of expressions during the evolutionary process, and improvement of the interpretability of symbolic models.

CCS CONCEPTS

• **Computing methodologies** → **Genetic programming**.

KEYWORDS

symbolic regression, genetic programming, equality saturation, simplification

ACM Reference Format:
Fabrício Olivetti de França and Gabriel Kronberger. 2023. Reducing Overparameterization of Symbolic Regression Models with Equality Saturation. In *Genetic and Evolutionary Computation Conference (GECCO '23), July 15–19, 2023, Lisbon, Portugal.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3583131.3590346

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
GECCO '23, July 15–19, 2023, Lisbon, Portugal
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0119-1/23/07...$15.00
https://doi.org/10.1145/3583131.3590346

1 INTRODUCTION

Regression analysis is the study of the relationship between independent and dependent variables of a phenomenon of interest described by a mathematical function, also known as regression model [9, 11]. The regression model can be used to make inferences for unseen data or to gain deeper insights on the studied phenomena [8]. In regression analysis one starts from a fixed function form $f(x, \theta)$, for input data x, and adjusts the values of θ to best fit the data. Different from this approach, genetic programming (GP) can be used for symbolic regression (SR) [13], searching for the best expression inside a search space composed of different mathematical expressions. The values of parameters (i.e., θ) are defined as a fixed part of the expression. The choice of the correct numerical parameters is a recurrent challenge in GP [21]. In early works, a limited set of ephemeral random constants (ERC) were initialized randomly and used as terminals together with input variables. Researchers relied explicitly on the power of GP to evolve the sequence of instructions to calculate the required values from the ERC and using the operations available in the function set. Since the function form is determined during the search, GP can generate overparameterized functions with redundant parameters. This redundancy may lead to undesirable effects because it impairs the interpretability of the expression and makes it difficult to calculate confidence intervals for parameter estimates, prediction intervals, or marginal likelihoods.

Many modern GP variants use a memetic approach [19] and search for a parameterized regression model $f(x, \theta)$ where the values of θ are adjusted before the evaluation. This approach was shown to increase the accuracy of the generated models rivaling with opaque machine learning models [12, 16]. However, it does not prevent overparameterized expressions which are assembled by crossover and mutation. If anything, the overparameterization slows down memetic approaches, because the more parameters we have, the larger the search space, effectively making the search more difficult. Also, an excess of parameters can cause multicolinearity of parameters that implies infinitely many solutions of the same goodness-of-fit. This can cause numerical issues and slower convergence, for example, when optimizing parameter values using iterative non-linear least squares algorithms [14].

One way to deal with this problem is to simplify expressions to eliminate redundant parameters and to improve the

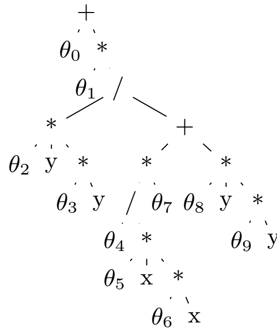

Figure 1: An overparameterized expression tree produced by Operon with input variables: x, y and parameters: θ.

parameterization of the problem. Common simplification algorithms [4] such as those implemented in Sympy [18] may not suffice, because the objective-function used for this simplification is for a very specific concept of what a simpler expression means, not necessarily to reduce the number of fitting parameters. Instead, we propose to use equality saturation [10, 27] (EqSat), a rewriting strategy that applies all the simplification rules keeping all the alternative function forms in an optimized data-structure called e-graph. This algorithm allows us to define a customized cost function that describes what *simple* means in our context and allows us to return expressions without redundant parameters which are suited best for the estimation of parameters.

This paper shows the benefits of EqSat for removing redundant parameters in SR models generated by different GP systems. We only analyze the effects on the final models and leave analysis of simplification of solution candidates within a GP run for future work. The expressions found with EqSat are compared to those simplified by a traditional approach (Sympy) and without any form of simplification.

The paper is organized as follows: in Section 2 we expand on the problem of overparameterization in GP. The EqSat data structures and algorithms are described in Section 3 with more detail. Sections 4 and 5 describe the experimental methodology followed by the discussion of the results. Finally Section 6 concludes this paper with some perspective on the use of EqSat in future research.

2 OVERPARAMETERIZATION IN GENETIC PROGRAMMING

Recently Kronberger [14] identified the problem of overparameterization, specifically when using Operon [3], a modern tree-based GP implementation for SR. The author noted that the models produced by Operon were usually overparameterized and mentioned a potential solution which is to simplify expressions such that the redundant parameters are merged together.

The example given in [14] is shown in Figure 1. The corresponding expression is

$$\theta_0 + \theta_1 \frac{\theta_2 y \theta_3 y}{\frac{\theta_4}{\theta_5 x \theta_6 x}\theta_7 + \theta_8 y \theta_9 y}, \quad (1)$$

which is obviously overparameterized with ten parameters. This means that the values of e.g. θ_2 and θ_3 cannot be identified uniquely. Some of the redundant fitting parameters in Equation 1 are introduced because Operon forces a multiplicative coefficient for each input variable and additionally scales all expressions linearly (θ_0, θ_1). However, it is obvious that other GP implementations may generate similar expressions. We will study other GP implementations in more details in Section 4.

Automatic algebraic simplification of the expression (e.g. using Sympy) leads to the simplified form

$$\theta'_0 + \frac{\theta'_1 y^2}{\theta'_2 y^2 + \frac{\theta'_3}{x^2}} \quad (2)$$

with only four parameters. However, this form is still overparameterized and can be further simplified to:

$$\theta''_0 + \frac{\theta''_1 y^2}{y^2 + \frac{\theta''_2}{x^2}}. \quad (3)$$

The additional parameters in Equation 1 only increase complexity but do not allow a better fit than the form with only three parameters in Equation 3 because the number of effective degrees of freedom of both expressions is the same. The only advantage could be that the original tree could be more evolvable because it provides more extension points to improve the fit.

This example already shows that simplification can be readily performed by computer algebra libraries, such as Sympy, but there is the limitation that such simplification does not necessarily lead to the expression with minimal number of fitting parameters. For example, the order in which rewriting rules are applied can have an effect on the final result. If we have the expression:

$$\frac{a(bx + cy)}{d}, \quad (4)$$

where a, b, c, d are fitting parameters and x, y are the input variables. If our set of rules is the following:

$$a(b + c) \rightarrow ab + ac \quad (5)$$
$$ab \rightarrow ba \quad (6)$$
$$(ab)/d \rightarrow a(b/d). \quad (7)$$

and the rules are to be applied in order, whereby operations where both arguments are fitting parameters are merged to a single parameter, the simplification steps would generate:

$$\frac{a(bx + cy)}{d} = \quad (8)$$
$$\frac{abx + acy}{d} = \quad (9)$$
$$\frac{a'x + b'y}{d}, \quad (10)$$

where a', b' are merged parameters. At this point, no other rule can be applied. Now, if we apply the rule in order 2, 3, 2, 1 instead we would arrive at the expression $(a'x + b'y)$ that contains only two parameters instead of three. One way to find the optimal order of rewrite rules is using EqSat [27] as it will be explained in the next Section 3.

3 EQUALITY SATURATION

Equality saturation is a technique applied to compiler optimizations and program synthesis [10, 20] that uses the data structure called *equality graph* (e-graph) to compactly represent all equivalent programs generated by the repeated application of rewrite rules. Given an initial program p, the general idea is to apply any applicable rule to this program storing all the generated equivalent programs in the structure. We repeat these steps until the e-graph is not changed, thus reaching the fixed point or saturation. When the e-graph is saturated, it means that it represents all possible equivalent programs for that specific set of rules and then we can extract a single program from this graph that minimizes a cost function. Contrary to the simplification process described above that destroys the current expression at every step, equality saturation only adds new information to the graph.

In the following we will use lowercase letters starting with a to match any expression tree (as a wildcard token). A more complex pattern can be described as an expression of different wildcards. So the pattern $a + b$ will match any tree with the operator $+$ as the root and with any two child trees. The pattern $a + a$ will match a tree with $+$ as the root but with the constraint that both left and right children must be exactly the same. A rewriting rule describes a pattern that must be matched and how to rewrite such pattern. For example, the rule $a+a \rightarrow 2*a$ says that every time we match the pattern $a + a$ we can replace it with $2 * a$.

The e-graph is composed of a set of e-classes, each e-class contains one or more e-nodes and each e-node represents a symbol (variable, constant, or function). The edges of this e-graph connects one e-node to an e-class with the property that any e-node must have exactly as many outgoing arrows as its arity and any e-class can have zero or more incoming arrows. The e-class without any incoming arrow is the root e-class and any e-node without outgoing arrows is a leaf e-node; they mark the start and the end of a traversal. We can extract an expression by traversing the e-graph starting from any e-node of the root e-class and recursively traversing each branch until it reaches a leaf. For example, refer to the left e-graph in Fig. 2 where the e-classes are represented by dashed boxes and the e-nodes by solid boxes. Starting from the e-node $+$ we can traverse the graph forming the expression $(1 * x) + x$.

This same figure illustrates the process of applying rewriting rules to update the e-graph. Let us suppose our rules set contains only the rules $1 * a \rightarrow a$ and $a + a \rightarrow 2 * a$. The only pattern that matches in the left e-graph is $1 * a$ for $a = x$, this rule creates a new e-node x in the e-class containing $*$ indicating that any traversal departing from this e-class will be equivalent to x. In this situation, the original e-class containing another e-node x is merged to this one, resulting in the middle e-graph. This new e-graph can represent infinitely many expressions such as $x+x, (1*x)+x, (1*(1*x))+x, \ldots$. Finally, from the middle e-graph we can apply the first rule again, resulting in the same e-graph, and the second rule, creating two new e-nodes, a $*$ inside the first e-class and a 2 that belongs to a new e-class. In this case, both $*$ e-nodes cannot be merged as they are not equivalent. This last e-graph can now represent the expressions $2*x, 2*(1*x), 2*(1*(1*x)), \ldots$ besides the previously expressions already represented in the middle e-graph.

This figure illustrates the invariance property of the e-graph stating that, any expression extracted departing from a given e-class will be equivalent.

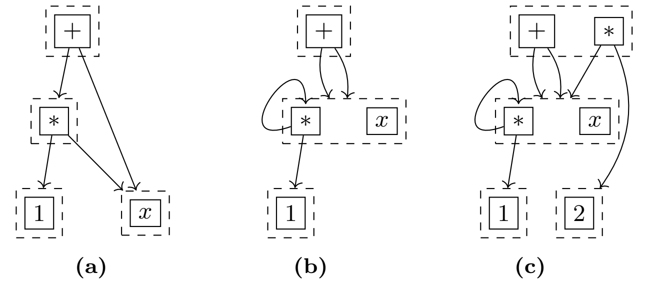

Figure 2: (a) Original expression; (b) after applying the rule $1 * x \rightarrow x$; (c) after applying the rule $x + x \rightarrow 2 * x$.

Since we keep all the generated equivalent expressions on the e-graph, we explore all possible orders of applying the set of rules. This procedure has two caveats that should be addressed for practical use. First of all, even though the e-graph compacts the storage of the equivalent programs, depending on the expression and set of rules, the e-graph can become excessively large or it may take a large amount of time to reach saturation. This is often solved by imposing a runtime, memory, and iteration limit. Another thing that should be avoided is the use of rules that can create cycles or induce an infinite sized expression.

In this paper, we use the Haskell library *hegg*[1] based on the Rust library *egg*[2] to apply the simplification process. These libraries implement optimized algorithms to build and maintain e-graph invariants [10] and to query an e-graph for a specific pattern [28]. These implementations also support a scheduler that limits the number of rules applied at every iteration and creates a tabu list to avoid repeated application of the same rule in the same sub-expressions.

4 EXPERIMENTS

With the following experiments we demonstrate that several well-known GP systems produce overparameterized SR models and that the number of parameters can be greatly

[1] https://hackage.haskell.org/package/hegg-0.3.0.0
[2] https://docs.rs/egg/latest/egg/

reduced using EqSat with the proposed rule set, whereby Sympy simplification is not sufficient. We chose two benchmark functions in which the generating function is non-linear and six open-source GP implementations. We have limited the detailed analysis of all GP systems to two benchmark functions because the exact nature of the data only has a small effect on the overparameterization issue. To demonstrate the effects occur frequently we additionally performed an analysis of Operon for a large set of benchmark instances.

We repeated the execution of each algorithm to each dataset 30 times storing the best expression found in each run. For each one of those expressions, we generated three simplified expressions by: i) applying EqSat to the original expression, ii) applying Sympy simplification algorithm to the original expression, and iii) applying EqSat to the expression simplified using Sympy. This last simplified expression was generated to test whether applying a destructive simplification algorithm creates a better initial point for the EqSat. As a result, we obtained four expressions per run, the original and the three simplified versions.

The comparison of the expressions of the 30 runs for a benchmark-algorithm combination forms a single independent experiment, totaling twelve experiments. These experiments will provide evidence to how much this problem affect these algorithms and how much the number of fitting parameters can be reduced. A broader study of how widespread this problem is for different combinations of algorithm and dataset is left for a future research.

We chose the top five algorithms (EPLEX, FEAT, GP-GOMEA, Operon, SBP) from SRBench [16], a recent benchmark of SR algorithms, and two additional recent implementations that reported competing results (Bingo, PySR), briefly described below. From this initial selection we removed FEAT due to a current bug[3] that returns an ill-formed symbolic expression. All of these algorithms, except for GP-GOMEA, perform nonlinear optimization of the adjustable parameters and Bingo also performs simplification with Sympy before adjusting the parameters.

4.1 GP Implementations

Many recent SR algorithms apply nonlinear optimization to determine the optimal parameters of a given expression. For example, *FEAT* [17] creates a map of the original input space to a transformed space by evolving multiple expression trees. These trees are combined using ridge regression and the inner coefficients of the tree are adjusted using backpropagation for every differentiable subtree. This is also known as Multiple Regression GP [1], when the algorithm produces a set of smaller trees combining them with a linear regression. Notice that if the algorithm does not allow parameters inside these trees, they will not create redundant parameters such as in [6]. *EPLEX* [15] stands out from the chosen algorithms because it uses an epigenetic representation and ϵ-lexicase selection, the parameter values are generated at random and optimized using a hill-climbing algorithm.

[3] https://github.com/cavalab/feat/issues/287

GP-GOMEA [26] generates parameter values during the initialization procedure and uses those throughout the iterations with the recombination operators. Even though this approach does not use parameter optimization, it is still capable of returning models with competitive accuracy and with a small size. As already mentioned, overparameterization may happen even without the use of parameter optimization.

Operon [3] implements nonlinear optimization for the numerical parameters with the Levenberg-Marquardt approach. It tends to use many parameters in expressions because it enforces a multiplicative coefficient for each variable terminal node and always adds two parameters for linear scaling of the whole expression.

More recently, a GP implementation in Julia with a Python interface, called *PySR* [5] (based on the Julia module *SymbolicRegression.jl*), was proposed with the goal of finding equations describing physical and engineering phenomena. The main goal of this framework is to provide an efficient and customizable SR implementation including constraints on unwanted constructs (such as nesting of nonlinear functions). This implementation can also use parameter optimization with either the BFGS [7] or Nelder-Mead methods.

Semantic backpropagation genetic programming (*SBP-GP*) uses semantically-aware operators to select the affected subtrees that behave most similarly. This technique applies affine transformations to improve the recombination operators and uses backpropagation to adjust the coefficients.

Bingo [23] was recently introduced as another Python framework for GP capable of customization of different parts of the algorithm. This framework provides a wrapper for the fitness function so that the user can apply any parameter optimization that is more adequate for the problem. In [2] Bayesian methods are used to optimize the parameters and quantify their uncertainties. One interesting novelty of this approach is that it simplifies the expression, using the algorithm described in [4], before adjusting the parameters.

4.2 Datasets

As for the datasets, we chose two benchmark functions in two variables Pagie-1 [22] (f_1) and Kotanchek [24] (f_2) for the detailed analysis of all GP systems:

$$f_1(x,y) = \frac{1}{1+x^{-4}} + \frac{1}{1+y^{-4}} \tag{11}$$

$$f_2(x,y) = \frac{e^{-(x-1)^2}}{1.2+(y-2.5)^2}. \tag{12}$$

Additionally, we analysed a huge set of models generated with Operon for all instances from the Feynman SR benchmark set [25].

As in the original papers, we have generated noiseless datasets with 1676 samples with $x, y \in [-5, 5]$ for f_1 and 2125 samples with $x, y \in [-0.2, 4.2]$, for f_2. We used the entire set for training the model[4] and executed each algorithm 30 times with different random seeds. We also used the default

[4] We do not use a test set because our objective is to analyze the number of parameters of SR models and not the goodness-of-fit of the

Table 1: Set of unconstrained rules used in this experiment.

$a+b \to b+a$	$(a*b)/c \to a*(b/c)$
$a*b \to b*a$	$a-0 \to a$
$a+(b+c) \to (a+b)+c$	$1*a \to a$
$a*(b*c) \to (a*b)*c$	$0/a \to 0$
$a*(b/c) \to (a*b)/c$	$a-a \to 0$
$0+a \to a$	$(a*b)+(a*c) \to a*(b+c)$
$0-a \to -a$	$a-(b+c) \to (a-b)-c$
$0*a \to 0$	$a-(b-c) \to (a-b)+c$
$-(a+b) \to -a-b$	$(1/a)*(1/b) \to 1/(a*b)$
$a-((-1)*b) \to a+b$	$a+(-b) \to a-b$
$\log(\exp a) \to a$	

hyperparameters for each algorithm with a common set of values of 200 generations, population size of 500, maximum tree length of 25 (when this choice was available), and the operators set $\{+,-,*,\div,\exp,\log\}$.

4.3 EqSat Rules

To minimize the number of fitting parameters, we have used a set of rules used for algebraic manipulation and additional rules that aim at merging parameters. Notice that before any rule is applied, it is assumed that any function applied to parameter values will be replaced by the evaluated value. So, we do not keep expressions such as $\exp(2+4)$ in the e-graph and simply reduce it to its evaluated value.

The simplification rules are in the form $e_1 \to e_2$ where e_1 is an expression of a pattern that must be matched and e_2 its replacement. For example, $a+b \to b+a$ will match any expression tree with the operator $+$ in its root and swap the left and right children. The rule $0*x \to 0$ will match any expression rooted at $*$ and which the left child is the tree 0 and the right child is any tree and simply replace it with 0. Whenever the lowercase letters may match any subtree, we will call it *unconstrained rules*, and they are reported in Table 1.

Some rules can only be applied if the left hand pattern abides to a certain condition. For example, the rule $\log(ab) \to \log(a)+\log(b)$ can only be applied if both a,b are positive. These are called constrained rules and the constraints are described as $\log(ab) \to \log(a)+\log(b), \forall a,b>0$ for this same example. These are called *constrained rules* and the set used in this paper is reported in Table 2.

The last set of rules have a condition that some of the pattern variables must be constant values (e.g., the fitting parameters). These rules, called *parameters rules*, are reported in Table 3 and the variables that must be numerical values are named a,b,c,d and the subtree variables are x,y.

models. This is the reason why we are not concerned about the error on an independent test set at the current stage of this research.

Table 2: Set of constrained rules used in this experiment.

$a*(1/a) \to 1, \forall a \neq 0$
$a/a \to 1, \forall a \neq 0$
$\log(a^b) \to b*\log a, \forall a,b>0$
$\log(a*b) \to \log a + \log b, \forall a,b>0$
$\log(a/b) \to \log a - \log b, \forall a,b>0$
$\exp(\log a) \to a, \forall a>0$

Table 3: Set of parameters specific rules used in this experiment. In this set of rules the variables a,b,c,d represents numerical values.

$(a*x)*(b*y) \to (a*b)*(x*y)$
$a*x+b \to a*(x+(b/a))$
$a*x-b \to a*(x-(b/a))$
$b-a*x \to a*((b/a)-x)$
$a*x+b*y \to a*(x+(b/a)*y)$
$a*x-b*y \to a*(x-(b/a)*y)$
$a*x+b/y \to a*(x+(b/a)/y)$
$a*x-b/y \to a*(x-(b/a)/y)$
$a/(b*x) \to (a/b)/x$
$x/(b*y) \to (1/b)*x/y$
$x/a+b \to (x+(b*a))/a$
$x/a-b \to (x-(b*a))/a$
$b-x/a \to ((b*a)-x)/a$
$x/a+b*y \to (x+(b*a)*y)/a$
$x/a+y/b \to (x+y/(b*a))/a$
$x/a-b*y \to (x-(b*a)*y)/a$
$x/a-b/y \to (x-y/(b*a))/a$
$(b+a*x)/(c+d*y) \to (a/d)*(b/a+x)/(c/d+y)$
$(b+x)/(c+d*y) \to (1/d)*(b+x)/(c/d+y)$
$(x-a) \to x+(-a)$
$(x-(a*y)) \to x+(-a*y)$

4.4 Implementation Details

To avoid running EqSat indefinitely, the used library implements a scheduler that limits the number of operations performed at every iteration to 2500. The process is repeated for 30 iterations and at every iteration it prioritizes the rules not applied in previous iterations. As this limitation may prevent reaching saturation, we do not have a guarantee that all of the equivalent expressions will be represented on the final e-graph. As such, we apply EqSat twice, first on the original expression and next to the result of the first application. We will see in the next section that these two steps approach is enough to obtain the desired effect without increasing the computational cost.

The cost function was designed to stimulate the minimization of the number of parameters by assigning a higher cost (5) to nodes for fitting parameters and a lower cost (1) to any other node, as illustrated in Alg. 1. This unitary cost is necessary to ensure we extract, in the first step, the expression

Algorithm 1 Cost function used to retrieve the equivalent expression with fewest parameters.

```
1:  function Cost(node)
2:      switch node do
3:          case op(a, b)
4:              return Cost(a) + Cost(b) + 1
5:          case f(a)
6:              return Cost(a) + 1
7:          case var
8:              return 1
9:          case const
10:             return 5
11: end function
```

with the most opportunity for re-parameterization in the next step. Take as an example the expression $(1/(c_1 x)) * (1/(c_2 x))$, where c_1, c_2 are fitting parameters. And suppose that the rule $(1/a) * (1/b) \rightarrow 1/(a * b)$ is applied in the last iteration of the first step because of the scheduler limitations. Since with all things being equal w.r.t. number of parameters, the algorithm will return the expression with the least number of nodes, when we apply EqSat again in the next step, it will be able to merge these redundant parameters. A command line interface to EqSat implementing these rules is available at https://github.com/folivetti/srtree-eqsat supporting expressions generated by all of the tested algorithms and some others not reported in this paper.

5 RESULTS

We can see the relative reduction in the number of parameters for the Pagie-1 benchmark obtained with the different simplification approaches for each algorithm in Fig. 3. In this plot we can see that EqSat obtained models with up to 50% fewer parameters. The reduction rate was larger on Operon, EPLEX, and PySR. For Bingo and SBP the number of parameters was reduced by 10% at most. GP-GOMEA expressions could not be reduced in most of the cases. On the other hand, simplifying expressions with Sympy led to more parameters (values below the horizontal line) in some cases. The only observable and consistent improvement was obtained for the expressions generated by PySR. The combination of Sympy and EqSat improved the results obtained with all but GP-GOMEA and SBP.

A comparison between approaches is reported in Fig. 4, where the left plot shows how much better EqSat is compared to Sympy and the right plot shows how much better EqSat is compared to the combination of both. From this plot we can see that EqSat often generates models with fewer parameters than Sympy alone, while the combination of both approaches produce consistently better results only for PySR.

Now, regarding Kotanchek function, we can see the ratio of reduction in Fig. 5. From this figure we can see that the application of EqSat always reduces the number of parameters, even for GP-GOMEA that had a rate of 0% for the Pagie function. The opposite is observed when using Sympy.

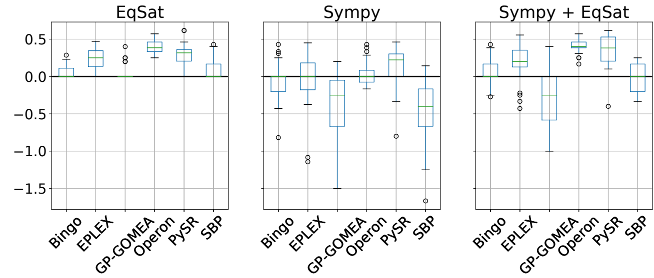

Figure 3: Boxplot of the ratio of decrease in number of parameters of the expressions using EqSat (left), Sympy (middle), and Sympy + EqSat (right) for the Pagie-1 function. Values above the bold line indicate a parameter reduction.

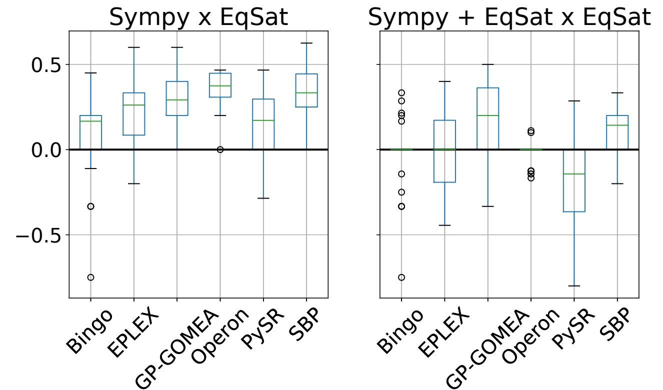

Figure 4: Boxplot of the ratio of decrease in number of parameters of the expressions using EqSat versus Sympy (left), and Sympy + EqSat versus Sympy (right) for the Pagie-1 function. Values above the bold line indicate that EqSat was more competent in reducing parameters.

We can see from this plot that it increases the number of parameters for most algorithms. Finally, there is no advantage when using the combination of both approaches, even though it does improve upon the Sympy results. This is confirmed by looking at Fig. 6 where we can see that EqSat is consistently better than Sympy alone and the combination of both.

To check whether EqSat succeeds in removing all linearly redundant parameters we compare the number of parameters after simplification to the numeric rank of the Jacobian matrix $J(x, \theta) = \left(\frac{\partial f(x,\theta))}{\partial \theta_1}, \ldots, \frac{\partial f(x,\theta))}{\partial \theta_k}\right)$ of the model. To determine the numeric rank we use the singular value decomposition of $J(x, \theta)$ as described in [14]. The numeric rank of the Jacobian matrix gives the number of linearly independent parameters in the function and therefore allows to check whether EqSat with the given rule set is capable to merge all linearly dependent parameters. The results shown in Figure 7 demonstrate the effectiveness of EqSat as the

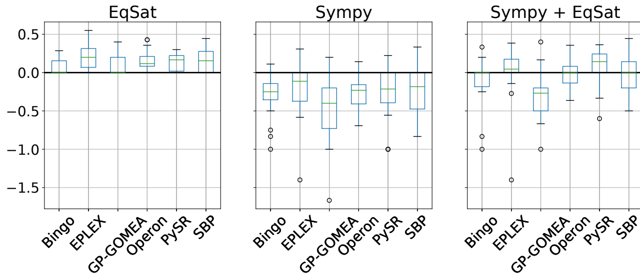

Figure 5: Boxplot of the relative reduction of the number of parameters using EqSat (left), Sympy (middle), and Sympy followed by EqSat (right) for the Kotanchek-1 function. Values above the bold line indicate a decrease in the number of parameters.

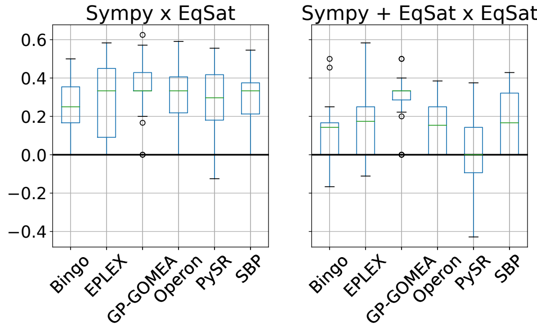

Figure 6: Boxplot of the relative reduction of the number of parameters using EqSat versus Sympy (left), and Sympy followed by EqSat versus Sympy (right) for the Kotanchek function. Values above the bold line indicate that EqSat was more competent in reducing parameters.

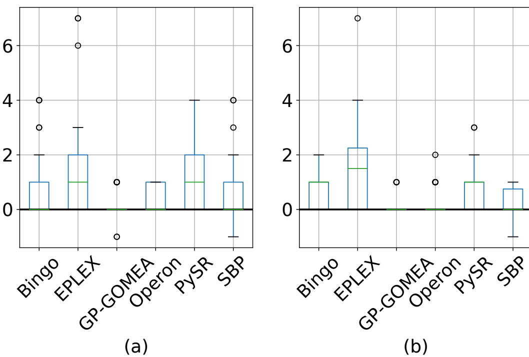

Figure 7: Difference between the number of parameters after EqSat simplification and the numeric rank for (a) Pagie-1 and (b) Kotanchek benchmarks.

boxplot of the difference between the number of parameters and the rank for the simplified expression with EqSat. For

Table 4: Percentage of the expressions in which EqSat reduced the number of parameters to the same value of the rank ($\Delta = 0$) or with at most one extra parameter ($\Delta \leq 1$). The expressions that originally met those conditions were removed from the count.

Algorithm	Pagie-1	Kotanchek
$\Delta == 0$		
Bingo	27.78%	22.22%
EPLEX	28.00%	18.75%
GP-GOMEA	30.77%	76.47%
Operon	66.67%	74.07%
PySR	36.67%	34.48%
SBP	42.86%	60.87%
$\Delta \leq 1$		
Bingo	33.33%	66.67%
EPLEX	45.45%	37.50%
GP-GOMEA	100.00%	100.00%
Operon	100.00%	94.44%
PySR	52.00%	71.43%
SBP	60.00%	100.00%

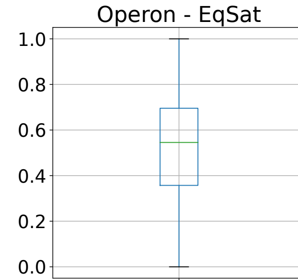

Figure 8: Relative reduction of the number of parameters using EqSat for 183491 different models generated by Operon using the Feynman benchmark data.

both datasets and all GP implementations the algorithm finds an expression with minimal number of parameters with a high probability. In most combinations, the median of this difference is 1, meaning that the simplified expression has one more parameter than the calculated rank. Table 4 summarizes the number of expressions in which EqSat achieve a certain difference from the numerical rank. From this table we can see that EqSat is most successful with Operon and SBP, possibly because how they envelope the nodes with redundant parameters. But, even other approaches that either use ERC, such as GP-GOMEA, or those that simplify the expression internally, such as Bingo, can benefit from EqSat reaching up to 76% of the ill-formed expressions properly repaired and up to 100% of those expressions with only a single extra parameter.

Finally, to verify whether this behavior is also observed with different datasets, we have generated 183491 models using Operon fitted on all SR Feynman benchmarks [25]

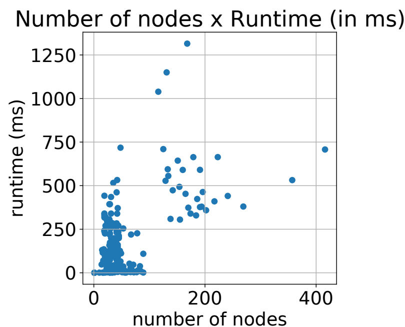

Figure 9: EqSat runtime of each generated expression for the Pagie-1 benchmark compared to the number of nodes of each expression.

with varying amount of noise added to the target. The boxplot in Fig. 8 depicts a similar behavior as observed in the two previous benchmark problems, evidencing that the overparameterization is innate to the algorithm rather dataset dependent.

Regarding the runtime, Fig. 9 shows the average time it takes to simplify the expression using EqSat for all the generated trees of the Pagie-1 benchmark. The x-axis represents the number of nodes and the runtime is measured in milliseconds. We can see from this plot that most of the expressions with a size in the range of 1 to 100 nodes require a runtime between $<$ 1ms and 400ms. For larger expressions (between 100 and 500 nodes) we observed a runtime between 300ms and 700ms. In a few cases, the runtime exceeded 1 second.

5.1 Discussion

From our experiments we can see that many GP algorithms generate expressions with an excessive number of parameters. Those implementations that use nonlinear optimization of parameters seems to suffer more from this problem. From the tested approaches, GP-GOMEA suffered the least with this problem, though it still produced redundant parameters. This is due to the fact that this algorithm prioritizes smaller solutions, than the other implementations. Bingo also produced solutions with just a few redundant parameters due to its internal simplification process.

Independent of the GP algorithm, EqSat consistently improved the solution whenever the original expression contained more parameters than its rank. It is worth noticing that this approach never produced worse solutions. On the other hand, Sympy was a hit or miss, often producing solutions with more parameters than the original. This is not surprising, because the Sympy simplification algorithm is not designed to minimize fitting parameters, but it is one of the few algebra libraries available for general use.

The runtime of EqSat can vary from less than 1ms to 1 second, depending on the characteristics of the expression. The average runtime is 178 ± 214 ms. With the average value, it would require 178 seconds for every 1000 evaluations. This is not an issue here, because we assume that only the best expressions returned by the algorithm are simplified. With a more conservative scheduler and with just a single step, the runtime could potentially be reduced to a reasonable time to be applied during the evolution process.

6 CONCLUSION

We investigated the problem of overparameterization in genetic programming (GP) for symbolic regression (SR) and how to merge the redundant parameters with equality saturation (EqSat). Overparameterization of SR models decreases the interpretability of the model. Different from common simplification algorithms, EqSat can maintain multiple equivalent expressions in its data structure and naturally explore the application of different sequences of rewrite rules in parallel.

To test whether EqSat is capable to remove redundant fitting parameters, we generated models using six GP implementations for two benchmark functions. We applied both EqSat and the Sympy simplification algorithm to those models and measured the decrease in the number of fitting parameters as well as the difference between the rank of the model and number of parameters.

The results showed that EqSat returned models with fewer fitting parameters for most of the tested GP implementations and, in the case that the original expressions contained no redundant parameters, it simply returned the same expression. In contrast, the Sympy simplification algorithm often increased the number of parameters, as it was designed with other objectives in mind.

Comparing with the numerical rank of the models Jacobian matrix, we can see that EqSat is capable of generating expressions with a number of parameters close to the numerical rank with high probability. In comparison, some algorithms produce expressions with up to seven more parameters than the rank.

For future research, we will evaluate the benefits of this simplification within three hypotheses: i) the nonlinear fitting on the simplified expression converges faster, ii) the nonlinear fitting is more robust to initialization, iii) the width of the confidence intervals of the parameters are reduced. We will also test whether applying EqSat on a selected set of solutions throughout the evolutionary process can lead to improved convergence.

ACKNOWLEDGMENTS

This research was funded by Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP), grant number 2021/12706-1. G.K. acknowledges support by the Christian Doppler Research Association, the Austrian Federal Ministry for Digital and Economic Affairs and the National Foundation for Research, Technology and Development within the Josef Ressel Center for Symbolic Regression.

REFERENCES

[1] Ignacio Arnaldo, Krzysztof Krawiec, and Una-May O'Reilly. 2014. Multiple regression genetic programming. In *Proceedings of the*

2014 Annual Conference on Genetic and Evolutionary Computation. 879–886.

[2] G. F. Bomarito, P. E. Leser, N. C. M. Strauss, K. M. Garbrecht, and J. D. Hochhalter. 2022. Bayesian Model Selection for Reducing Bloat and Overfitting in Genetic Programming for Symbolic Regression. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion* (Boston, Massachusetts) *(GECCO '22)*. Association for Computing Machinery, New York, NY, USA, 526–529. https://doi.org/10.1145/3520304.3528899

[3] Bogdan Burlacu, Gabriel Kronberger, and Michael Kommenda. 2020. Operon C++: An Efficient Genetic Programming Framework for Symbolic Regression. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion (GECCO '20)*. Association for Computing Machinery, internet, 1562–1570. https://doi.org/doi:10.1145/3377929.3398099

[4] J.S. Cohen. 2018. *Computer Algebra and Symbolic Computation: Mathematical Methods*. CRC Press. https://books.google.at/books?id=0WO2zQEACAAJ

[5] Miles Cranmer. 2020. *PySR: Fast & Parallelized Symbolic Regression in Python/Julia*. https://doi.org/10.5281/zenodo.4041459

[6] Fabrício Olivetti de França and Guilherme Seidyo Imai Aldeia. 2021. Interaction–Transformation Evolutionary Algorithm for Symbolic Regression. *Evolutionary computation* 29, 3 (2021), 367–390.

[7] Roger Fletcher. 2013. *Practical methods of optimization*. John Wiley & Sons.

[8] Andrew Gelman, Jennifer Hill, and Aki Vehtari. 2020. *Regression and other stories*. Cambridge University Press.

[9] Frank E Harrell. 2017. Regression modeling strategies. *Bios* 330, 2018 (2017), 14.

[10] Rajeev Joshi, Greg Nelson, and Keith Randall. 2002. Denali: A goal-directed superoptimizer. *ACM SIGPLAN Notices* 37, 5 (2002), 304–314.

[11] Robert E Kass. 1990. Nonlinear regression analysis and its applications. *J. Amer. Statist. Assoc.* 85, 410 (1990), 594–596.

[12] Michael Kommenda, Bogdan Burlacu, Gabriel Kronberger, and Michael Affenzeller. 2020. Parameter identification for symbolic regression using nonlinear least squares. *Genet. Program. Evolvable Mach* 21, 3 (2020), 471–501.

[13] John R. Koza. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA. http://mitpress.mit.edu/books/genetic-programming

[14] Gabriel Kronberger. 2022. Local Optimization Often is Ill-conditioned in Genetic Programming for Symbolic Regression. *arXiv preprint arXiv:2209.00942* (2022).

[15] William La Cava and Jason H Moore. 2019. Semantic variation operators for multidimensional genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference*. 1056–1064.

[16] William La Cava, Patryk Orzechowski, Bogdan Burlacu, Fabricio Olivetti de Franca, Marco Virgolin, Ying Jin, Michael Kommenda, and Jason H. Moore. 2021. Contemporary Symbolic Regression Methods and their Relative Performance. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*. https://openreview.net/pdf?id=xVQMrDLyGst

[17] William La Cava, Tilak Raj Singh, James Taggart, Srinivas Suri, and Jason H Moore. 2018. Learning concise representations for regression by evolving networks of trees. *arXiv preprint arXiv:1807.00981* (2018).

[18] Aaron Meurer, Christopher P Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K Moore, Sartaj Singh, et al. 2017. SymPy: symbolic computing in Python. *PeerJ Computer Science* 3 (2017), e103.

[19] Pablo Moscato. 1999. Memetic Algorithms: A Short Introduction. In *New Ideas in Optimization*, David Corne, Marco Dorigo, and Fred Glover (Eds.). McGraw-Hill, London, 219–234.

[20] Chandrakana Nandi, Max Willsey, Adam Anderson, James R Wilcox, Eva Darulova, Dan Grossman, and Zachary Tatlock. 2020. Synthesizing structured CAD models with equality saturation and inverse transformations. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 31–44.

[21] Michael O'Neill, Leonardo Vanneschi, Steven Gustafson, and Wolfgang Banzhaf. 2010. Open issues in genetic programming. *Genetic Programming and Evolvable Machines* 11, 3 (01 Sep 2010), 339–363. https://doi.org/10.1007/s10710-010-9113-2

[22] Ludo Pagie and Paulien Hogeweg. 1997. Evolutionary Consequences of Coevolving Targets. *Evolutionary Computation* 5, 4 (Winter 1997), 401–418. https://doi.org/doi:10.1162/evco.1997.5.4.401

[23] David L Randall, Tyler S Townsend, Jacob D Hochhalter, and Geoffrey F Bomarito. 2022. Bingo: a customizable framework for symbolic regression with genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. 2282–2288.

[24] Guido Smits and Mark Kotanchek. 2004. Pareto-Front Exploitation in Symbolic Regression. In *Genetic Programming Theory and Practice II*, Una-May O'Reilly, Tina Yu, Rick L. Riolo, and Bill Worzel (Eds.). Springer, Ann Arbor, Chapter 17, 283–299. https://doi.org/doi:10.1007/0-387-23254-0_17

[25] Silviu-Marian Udrescu and Max Tegmark. 2020. AI Feynman: A physics-inspired method for symbolic regression. *Science Advances* 6, 16 (2020), eaay2631.

[26] Marco Virgolin, Tanja Alderliesten, Cees Witteveen, and Peter AN Bosman. 2017. Scalable genetic programming by gene-pool optimal mixing and input-space entropy-based building-block learning. In *Proceedings of the Genetic and Evolutionary Computation Conference*. 1041–1048.

[27] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. egg: Fast and extensible equality saturation. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–29.

[28] Yihong Zhang, Yisu Remy Wang, Max Willsey, and Zachary Tatlock. 2022. Relational e-matching. *Proceedings of the ACM on Programming Languages* 6, POPL (2022), 1–22.