# Linear Scaling with and within Semantic Backpropagation-based Genetic Programming for Symbolic Regression

Marco Virgolin
Centrum Wiskunde & Informatica
Amsterdam, The Netherlands
Marco.Virgolin@cwi.nl

Tanja Alderliesten
Amsterdam UMC
University of Amsterdam
Amsterdam, The Netherlands
T.Alderliesten@amc.uva.nl

Peter A.N. Bosman
Centrum Wiskunde & Informatica
Amsterdam, The Netherlands
Delft University of Technology
Delft, The Netherlands
Peter.Bosman@cwi.nl

## ABSTRACT

Semantic Backpropagation (SB) is a recent technique that promotes effective variation in tree-based genetic programming. The basic idea of SB is to provide information on what output is desirable for a specified tree node, by propagating the desired root-node output back to the specified node using inversions of functions encountered along the way. Variation operators then replace the subtree located at the specified node with a tree for which the output is closest to the desired output, by searching in a pre-computed library. In this paper, we propose two contributions to enhance SB specifically for symbolic regression, by incorporating the principles of Keijzer's Linear Scaling (LS). In particular, we show how SB can be used in synergy with the scaled mean squared error, and we show how LS can be adopted within library search. We test our adaptations using the well-known variation operator Random Desired Operator (RDO), comparing to its baseline implementation, and to traditional crossover and mutation. Our experimental results on real-world datasets show that SB enhanced with LS substantially improves the performance of RDO, resulting in overall the best performance among all tested GP algorithms.

## CCS CONCEPTS

• **Computing methodologies** → **Machine learning**; **Genetic programming**;

## KEYWORDS

genetic programming, semantic backpropagation, linear scaling

**ACM Reference Format:**
Marco Virgolin, Tanja Alderliesten, and Peter A.N. Bosman. 2019. Linear Scaling with and within Semantic Backpropagation-based Genetic Programming for Symbolic Regression. In *Genetic and Evolutionary Computation Conference (GECCO '19), July 13–17, 2019, Prague, Czech Republic.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3321707.3321758


*GECCO '19, July 13–17, 2019, Prague, Czech Republic*

ACM ISBN 978-1-4503-6111-8/19/07…$15.00
https://doi.org/10.1145/3321707.3321758

## 1 INTRODUCTION

Semantic Backpropagation (SB) is a recent technique in tree-based Genetic Programming (GP) [14, 20] which enables the design of novel variation operators [19, 26]. For any tree node, given a target output for the tree, SB determines what the *desired output* for that node is. If the node were to be replaced with a subtree that delivers the desired output, then the outputs of the ancestor nodes would also change, ultimately making the root deliver the target output. The application of SB-based GP algorithms has been shown to be particularly effective in supervised learning applications such as Boolean circuit synthesis and symbolic regression [8, 15, 19].

SB-based variation operators modify trees by replacing nodes with subtrees that match desired outputs as closely as possible. The Random Desired Operator (RDO) is perhaps the most known among them, as it has been shown to perform best on a variety of problems [19, 26]. Key components of RDO are the use of a library of trees with pre-computed outputs, and a library search procedure to retrieve the tree which most closely matches the desired output.

As to the library, two traditional ways exist to build it [19]. The first way is to generate all possible trees within a maximum tree height, and to retain one tree for each unique output (the tree with less nodes is kept). Clearly, this method cannot scale with the number of dimensions, nor with the sampling of real-valued constants. In [19], for problems with a single feature, a maximum height of 3 already results in hundreds of thousands of trees. The second way is to dynamically refresh the library every generation, by including all subtrees with unique output as observed in the population. The downside of this approach is that the expressiveness of the library may be limited, as it is biased by how the population evolves.

Linear Scaling (LS) is an interesting existing technique to minimize the mean squared error of a GP tree by applying an optimal linear transformation to the output of the tree [12, 13]. While typically used to improve the fitness, LS can more generally be applied to scenarios where a (monotonic transformation of the Euclidean) distance between two outputs needs to be minimized. As SB-based GP operates by matching desired outputs, it stands to reason that some form of LS can be integrated to benefit the algorithm. This is precisely the topic of this paper: we study how to best integrate and how to best observe the impact of LS on SB-based GP.

We propose, for the first time, the use of LS as (i) A separate, but synergetic mechanism, to work *with* SB; and (ii) A joint mechanism, to use *within* SB-based GP, namely during library search.

Much previous work solely considered synthetic benchmark functions with few variables, and generational computation budgets (see Sec. 3). The latter choice arguably favors SB-based GP when compared to other forms of GP (e.g., traditional GP that swaps and mutates subtrees randomly [14]), in that the computational time taken by SB itself, library construction, and library search, is not considered [19]. For this reason, in our experiments, we assess the effectiveness of the proposed LS-enhanced, SB-based GP in terms of *both* number of generations and *time*. Moreover, we test the algorithms on realistic small- to medium-sized regression problems, using ten established real-world benchmark datasets.

## 2 SEMANTIC BACKPROPAGATION

Given a target output $\mathbf{t}$ for the tree (i.e., for its root), SB computes a desired output $\mathbf{d}^N$ for one specific node $N$. This information can then be used to replace $N$ with a subtree that has output as close as possible to $\mathbf{d}^N$. It is expected that, the closer the output of the subtree is to $\mathbf{d}^N$, the closer the output of the root will be to $\mathbf{t}$.

Let $D$ be the depth of $N$. Then $N$ has $D$ ancestors. Let $A_k$ be the ancestor of $N$ at depth $k$. Similarly, let $\mathbf{S}_k = \{S_k^1, S_k^2, \dots\}$ be the (possibly empty) set of sibling nodes of $A_k$. For the sake of brevity, we now use the same notation used to refer to a node to also identify the function implemented by that node. E.g., $A_k\ (\mathbf{x}, \mathbf{S}_{k+1})$ represents the application of the function of node $A_k$ on $\mathbf{x}$, and on (the outputs of) the nodes $\mathbf{S}_{k+1}$. Therefore, we can say that SB computes:

$$\mathbf{d}^{A_{k+1}} = A_k^{-1}\left(\mathbf{d}^{A_k}, \mathbf{S}_{k+1}\right), \tag{1}$$

where $A_k^{-1}$ represents the inversion of the function $A_k$. SB starts from the root by setting its desired output to the target, i.e. $\mathbf{d}^{A_0} := \mathbf{t}$. The recursive computation of the desired output for $N$ (at depth $D$) then follows: $\mathbf{d}^N = A_{D-1}^{-1}\left(\mathbf{d}^{A_{D-1}}, \mathbf{S}_D\right)$. Fig. 1 shows an example.

Note that, if non-injective functions (e.g., $abs(\cdot)$) are included in the function set, each desired output vector will grow to represent different possible outcomes, i.e., $\mathbf{d} \in \mathbb{R}^{\gamma \times n}$ and $\mathbf{d}_i = \{d_i^1, \dots, d_i^\gamma\}$, with $i = 1, \dots, n$ the indices of training examples (from now on, for brevity, we drop the superscript $N$ from $\mathbf{d}^N$). Note that $\gamma$ can be $\infty$, e.g., for sin. Similarly, any value may satisfy some inversions: e.g., for $x = \times^{-1}(0,0)$, $0 \times x = 0, \forall x \in \mathbb{R}$. In such cases, we will indicate that any value is good with $*$. We describe how $\mathbf{d}_i$ with multiple and/or $*$ values is handled during library search in the following Section 2.1. Conversely, some functions are not invertible (in $\mathbb{R}$) in some points (e.g., $(\cdot^2)^{-1}(-1) = \sqrt{-1}$), thus some $\mathbf{d}_i^j$ may not exist. If SB is unfeasible, i.e., $\exists i : \mathbf{d}_i = \emptyset$, we abort SB (as in [19]).

### 2.1 Library and library search

Given the desired output $\mathbf{d}$, a subtree with similar output is sought. For this purpose, typical SB-based variation operators rely on a searchable library of pre-computed trees with unique outputs. As aforementioned in the introduction, a way to build the library is to pre-compute all possible trees up to a maximum height, but this becomes intractable with already few features (terminals) [19]. The other typical method is to collect all subtrees as observed in the population (updated every generation) [4, 22].

In the so-called "population-based" library, if multiple subtrees with the same output exist in the population, the one smallest in terms of the number of nodes is retained. Furthermore, subtrees with constant output are not considered (library search handles constants separately, see below).

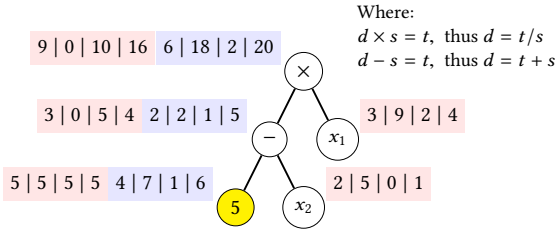


**Figure 1: Example of SB for the yellow leaf. The current outputs of each node are in pink. The desired outputs are in blue. The desired output of the root is the (given) target output, and the others are computed by recursive inversion on the path down to the yellow leaf. The operations in the top right describe the inversions used in this example.**

The library search procedure parses the library to find the tree of which its output $\mathbf{o}$ minimizes the distance from $\mathbf{d}$, e.g., in terms of *modifications* of the L1 or L2 distance (or any Minkowski distance). By modified version of the distances we mean that the multi-valued nature of $\mathbf{d}_i$ must be accounted for. The distance can be computed by finding the one $\mathbf{d}_i^j$ that minimizes $|\mathbf{d}_i^j - \mathbf{o}_i|^w, \forall w \in \{1, 2, \dots\}$ [19]. Furthermore, if $\exists j : \mathbf{d}_i^j = *$, then the values of $\mathbf{d}_i$ do not matter, and it is defined to have $|* - \mathbf{o}_i|^w = 0$. By pre-sorting the $\mathbf{d}_i^j$ in $j$, a linearly addressable library $\mathscr{L}$ can be parsed in $O(|\mathscr{L}|n \log \gamma)$ [19].

With trees with a constant output being typically excluded from the library, library search further considers the distance between a constant value and the desired output in a separate fashion. In [19], the values of $\mathbf{d}$ are considered to be candidate constant values, and the best one is picked. The best tree found in the library, or a single-node implementing the best constant, is finally returned by the library search procedure, depending on which is closest to $\mathbf{d}$.

### 2.2 Random Desired Operator

RDO works by generating an offspring tree that differs from the parent in one subtree. The pseudocode of RDO is shown in Algorithm 1. First, the offspring ($O$) is created as a clone of the parent ($P$), and one of its nodes $N$ is selected. In [19], it is proposed to select $N$ with the *equal depth probability* criterion, which first samples the depth $D$ to consider, and then samples $N$ among the nodes with depth $D$, both uniformly at random. Second, SB is executed for $N$, by setting the target for the root to the dependent variable to regress, i.e. $\mathbf{t} := \mathbf{y}$. Note that, in general, $\mathbf{t}$ can be different (e.g., a crossover operator is proposed in [15] that sets the target output for one parent to the output of another parent). If SB is aborted because an unfeasible $\mathbf{d}$ is computed, RDO returns the clone of the parent. Otherwise, library search is performed, resulting in a tree $T$ that has output with minimum distance from $\mathbf{d}$. Finally, RDO returns the offspring, adapted by replacing its subtree at $N$ with $T$.

### 2.3 Intermediate output caching

SB-based GP is particularly efficient if the output of subtrees are cached. In particular, each recursive iteration of SB requires to

**Algorithm 1** Pseudocode of RDO.

```
1 function RDO(y, P, 𝒟)
2     O ← Clone(P)
3     N ← EqualDepthProbability(O)
4     d ← SemanticBackpropagation(N, y)
5     if ∃i : d_i = ∅ then return O
6     T ← LibrarySearch(d, 𝒟)
7     O ← ReplaceSubtree(N, T)
8     return O
```

know the output of the sibling nodes, and library search requires the output of the library trees. Therefore, in [19] it is proposed to cache intermediate tree outputs, i.e., the outputs of all nodes.

Intermediate output caching not only speeds up SB-related methods, but also the traditional evaluation of trees. In fact, if a node is changed, it is sufficient to recompute the outputs only along the chain of ancestors, i.e., from the parent of that node upwards, to get the output of the root. While these *partial evaluations* can be very effective, especially for high-dimensional outputs, they take a toll in terms of memory (see, e.g., the discussion on scalability in [24]).

## 3 RELATED WORK

Two research lines are mostly related to this paper, namely the one on LS, and the one on SB. As to LS, the most cited work to date is [12], which shows how LS can dramatically improve the performance of GP, for synthetic functions with up to three variables. In [13], theoretical motivations for the added value of LS are given. LS was successfully used for practical applications in, e.g., [1, 21, 23].

To the best of our knowledge, no contribution has been made that proposes modifications of LS itself. This is not surprising, as LS is quick (i.e., $O(n)$), and the scaling and translation coefficients are optimal w.r.t. the dependent variable $\mathbf{y}$ (see the description of LS in Sec. 4.1). Perhaps more interestingly, we also could not find any work where LS is combined with another method in a truly synergetic way, i.e., having LS and/or the other method sharing information with each other. E.g., in [5], LS is used together with a particular mating scheme, but the two methods co-exist independently from each other. Here, we consider for the first time a use of LS that is deeply intertwined with another method, i.e., SB.

As to SB, it was first introduced together with RDO in [26], and much research work has followed. Perhaps one of the most comprehensive contributions is [19], which compares several variation operators, two of which are SB-based (RDO, and approximately geometric crossover [15]). RDO is shown to outperform most of the other operators, on both Boolean and regression problems.

While RDO uses SB to replace a subtree, the Forward Propagation Mutation (FPM) operator proposed in [22] does the opposite: it preserves the subtree, and replaces the remaining part of the tree, called the *context*. A new context is built by determining a new root, and another subtree to append to the root, which is a sibling to the preserved one. This new subtree is retrieved by library search using cosine similarity, and is rescaled by an optimal constant (determined in $O(n)$). The authors claim that an alternative could be to use LS to also determine a translation coefficient during library search for FPM. This is indeed investigated in our work, for RDO.

Very recently, a variant of FPM has been proposed in [3] where the target is set to a random point in the segment between $\mathbf{y}$ and the

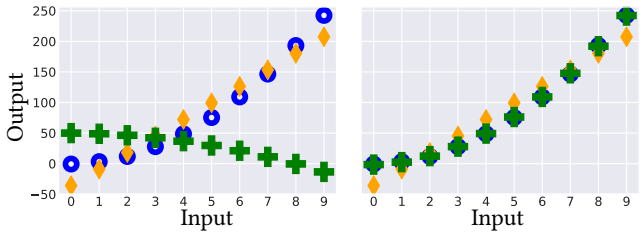


**Figure 2: Example of the effect of LS. Blue circles represent the output of the function to approximate, while orange diamonds and green crosses are the output of two trees. Left: The orange diamonds are closest to the blue circles. Right: The application of LS substantially improves the green crosses, making them become the best match.**

output of the parent, and where LS is applied *after* library search. While this improves the fitting of the subtree to the context, it is less effective (but faster) compared to considering optimal translation coefficients *during* library search (as recognized by [22]).

Notwithstanding the novelty and advantages of the aforementioned and of other work on SB-based GP (e.g., [8, 11]), as we mentioned in the introduction, mostly synthetic functions have been considered so far, in the domain of regression. These functions have up to three variables only. Furthermore, comparisons have only been framed in terms of number of generations, thus ignoring much of the computational expensiveness of SB-based GP.

To the best of our knowledge, only in [3] and [4] four and two real-world benchmarks are respectively considered, for GP using RDO (and a variant) and the aforementioned variant of FPM. Yet again, only generational budgets are considered, except for the supplementary material of [3], where experiments using a time limit are reported. Although those results undeniably bring additional insight, we believe it remains hard to assess what the impact of using SB-based operators is on computation time, because of two reasons. Firstly, a relatively small population size of 100 is used, meaning that population-based libraries will also be small and quick to parse. Secondly, the considered GP algorithms have several differences (e.g., selection schemes), and always employ other variation operators together with the SB-based ones.

In this paper, not only do we consider how LS can be combined with SB-based GP, but we also attempt to address the main limitations of the related work. We adopt ten real-world benchmark datasets for regression with dozens of features, as they are arguably more representative of practical problems, and we attempt to frame algorithmic comparisons in terms of both generations and time limits to also observe the potential overhead of adopting SB-based GP.

## 4 LINEAR SCALING *WITH* SB-BASED GP

We now describe the first contribution of this paper, i.e., how SB can work together with LS, in synergy. The main concept behind LS is to allow GP to focus on the "shape" of the function to approximate, by providing translation and scaling coefficients that minimize the training Mean Squared Error (MSE) [12] (see, e.g., Figure 2).

In principle, LS and SB can work independently, without making changes to the two methods. In RDO, SB works by setting the

target output $\mathbf{t}$ (i.e., the desired output for the root) to $\mathbf{y}$. To backpropagate this information means to directly try to optimize the tree towards delivering exactly $\mathbf{y}$, without exploiting the fact that LS helps scaling the output of the root. In other words, in this setting, LS acts solely as a "patch" on top of SB-based GP, as it attempts to correct for the residual error that the algorithm normally makes.

We argue that it is reasonable to attempt to make LS and SB work in synergy to reduce the error, in particular by informing SB on what the effect of LS is. Indeed, we hypothesize that if the transformation applied by LS is also backpropagated to determine the desired output, the subsequent variation will be more effective, as it will attempt to correct the error that remains *after* LS is applied.

### 4.1 Linear scaling

LS works as follows. Let the MSE between the dependent variable $\mathbf{y}$ and the tree output $\mathbf{o}$ be the fitness function for regression:

$$\text{MSE}(\mathbf{y}, \mathbf{o}) = \frac{1}{n}\sum_{i=1}^{n}(\mathbf{y}_i - \mathbf{o}_i)^2.$$

LS introduces a *scaled* version of the MSE, where respectively a translation coefficient $a$ and a scaling coefficient $b$ are used within the computation of the MSE, in order to minimize it:

$$\text{MSE}^{a,b}(\mathbf{y}, \mathbf{o}) = \frac{1}{n}\sum_{i=1}^{n}(\mathbf{y}_i - (a + b\mathbf{o}_i))^2.$$

The optimal $a$ and $b$ that minimize the error are (see [12]):

$$a = \bar{\mathbf{y}} - b\bar{\mathbf{o}},$$
$$b = \sum_{i=1}^{n}\frac{(\mathbf{y}_i - \bar{\mathbf{y}})(\mathbf{o}_i - \bar{\mathbf{o}})}{(\mathbf{o}_i - \bar{\mathbf{o}})^2} = \frac{cov(\mathbf{y}, \mathbf{o})}{var(\mathbf{o})}. \quad (2)$$

The implementation of Eq. 2 takes $O(n)$.

### 4.2 Linear scaling in synergy with semantic backpropagation

We now describe how LS and SB can work in synergy. To begin, we point out that using the $\text{MSE}^{a,b}$ is equivalent to using the traditional MSE on a tree where the addition of $a$ and multiplication by $b$ are encoded within the tree itself, with suitable nodes placed on top of the root. For example, consider the rightmost tree of Figure 3: ignore the nodes in white, and imagine the plus node to be the actual root. That tree is essentially one where the effect of LS is incorporated in its structure (with pink nodes). For such a tree, it is straightforward to compute SB (as described in Sec. 2). In particular, we immediately see that for a target output $\mathbf{t}$, the desired output for the original root (top green node) will be:

$$\mathbf{d}_i = \frac{\mathbf{t}_i - a}{b}, \forall i. \quad (3)$$

Therefore, whenever SB needs to be performed, we can calculate $a$ and $b$ based on $\mathbf{t}$ and the current tree output $\mathbf{o}$ (or, if $\mathbf{t} = \mathbf{y}$ like in RDO, $a$ and $b$ can be cached after they are computed for $\text{MSE}^{a,b}(\mathbf{y}, \mathbf{o})$). Then we can compute $\mathbf{d}$ for the root using Eq. 3 as starting point for SB, and then we can proceed with Eq. 1 as usual.

We remark that the computation overhead for including LS in SB this way can be considered negligible. If $D$ is the depth of the node chosen for replacement, then SB needs to compute $D$ inversions. If only injective functions are considered, this leads to $O(Dn)$ (it is $O(D\gamma^D n)$ if non-injective functions are present). In typical symbolic regression settings, $D$ is bounded by a small constant and $n$ is large, i.e. $D \ll n$, meaning that the bound is $O(n)$. Since to include LS in synergy with SB means to compute Eq. 2 and to compute Eq. 3, and since these computations bring only additional $O(n)$ contributions, the bound remains $O(n)$.

## 5 LINEAR SCALING *WITHIN* SB-BASED GP

When performing library search, a tree $T$ that has output $\mathbf{o}$ close to a desired output $\mathbf{d}$ is sought for. This situation is similar to the symbolic regression problem itself, where the output of the root node is expected to match $\mathbf{y}$. Because LS is known to help in the latter scenario [12, 13], it is reasonable to expect that LS can improve the effectiveness of library search as well, as optimally scaled tree outputs will be considered.

### 5.1 Linear scaling during library search

Let L2 be the distance metric adopted by the library search procedure. Since L2 is a monotonic transformation of the MSE, the optimal coefficients $a$ and $b$ can be computed with Eq. 2 (replacing $\mathbf{y}$ with $\mathbf{d}$) to decrease the distance between $\mathbf{d}$ and $\mathbf{o}$.

In practice, $\mathbf{d}$ needs to be in $\mathbb{R}^n$ (instead of in $\mathbb{R}^{\gamma\times n}$) to have a unique, well-defined way to compute $a$ and $b$. E.g., what is $\bar{\mathbf{d}}$ if there exists some $\mathbf{d}_i$ with multiple values? Some criterion should be used to choose one of the values for $\mathbf{d}_i$ (e.g., the value closest to the mean given by considering the other $\mathbf{d}_j$ that have unique values). Restricting the multi-dimensionality of the desired output by choosing a single value for each $\mathbf{d}_i$ means that possibly better matching outputs present in the library will not be searched for. Alternatively, multiple scalings could be computed and the best one could be taken, but exponential possibilities could exist. In this paper, we include a non-injective function in the function set of GP that is symmetric around zero. For its inversion, we choose to return only the positive value (see Sec. 6). Regarding $*$ values, we make the assumption that, if present, they are few, and can be ignored when computing $a$ and $b$.

We thus assess the effect of using LS within library search. Whenever library search is performed, for each tree in the library, we compute optimal $a$ and $b$ coefficients that minimize the distance between the output of the tree and the desired output. Library search then returns the best matching tree, along with its $a, b$ coefficients. When this tree needs to be appended by the variation operator, four nodes are added on top of its root, namely two constants with value $a$ and $b$ respectively, an addition and a multiplication node, to effectively incorporate the scaling in the structure of the tree. Figure 3 illustrates this procedure.

The computation time taken by LS is $O(n)$, and it is additive w.r.t. the time taken to compute the distance between the output of a tree in the library and $\mathbf{d}$. Therefore, the library search bound remains $O(|\mathscr{L}|n)$. In practice, some adjustments can be made to reduce computations. Once the library is created, the mean of each tree output $\bar{\mathbf{o}}$ can be cached, as well as the terms $(\mathbf{o}_i - \bar{\mathbf{o}})$ (see Eq. 2). Furthermore, the mean of the desired output $\bar{\mathbf{d}}$, and the terms $(\mathbf{d}_i - \bar{\mathbf{d}})$ can be computed only once, before starting the library search. This way, the only operation with cost linear in $n$ that is left to do when searching is the computation of the numerator of $b$ in Eq. 2.

Lastly, when using LS within library search, we also change the way a competing constant is computed: we set the constant to the optimal value, i.e., $\bar{\mathbf{d}}$.

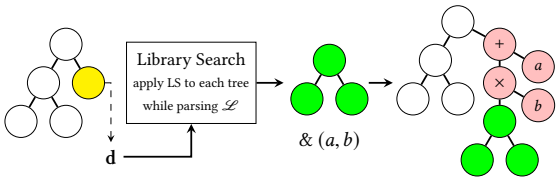


**Figure 3: Illustration of SB-based GP variation using LS within library search. From left to right: the desired output d is computed for a node (in yellow). Library search retrieves the tree (in green) with output that, scaled by $a$, $b$, best matches d. The yellow node is replaced, and a structure is incorporated to account for the scaling (in pink).**

## 6 EXPERIMENTAL SETUP

The parameter settings for GP are reported in Table 1, and are typical settings used in literature ([20], related work of Sec. 3). The function $\div_{\text{AQ}}$ is the analytic quotient [18], which allows for smooth division with no discontinuities (the denominator can never become 0). The inversions for the functions considered are reported in Table 2. Note that in the inversion of $\div_{\text{AQ}}$ for $a_j$, we return only the positive value. The terminal set includes an Ephemeral Random Constant (ERC) [20], that has the effect of generating nodes with randomized constant output. These constant outputs are sampled uniformly in the interval defined by the minimum and maximum value (available at training time) of the features.

Together with SB-based GP using RDO, we consider as a baseline GP with standard subtree crossover and subtree mutation operators (SGP) [14, 20], respectively applying them on 90% and 10% of the population every generation. Like for RDO, the nodes to swap/mutate are chosen with equal depth probability, as in [19].

The operators of SGP take much less computation time compared to RDO (essentially $O(1)$), in particular because the latter requires to build the library of trees, and performs SB and library search. Therefore, we consider both a limit of 100 generations and a time-dependent limit of 1000 seconds. As time-based comparisons can very much depend on implementation details, we attempt to boost their fidelity by developing all algorithms in the same C++ code base, which can be found at: https://goo.gl/UbFFSU.

We consider ten real-world benchmark regression datasets, with variable numbers of examples and features, as reported in Table 3. The datasets Dow chemical and Tower are recommended as benchmarks in [25]. The others are often used in GP literature and come from the UCI machine learning repository[1]. These datasets can be considered "well-behaved", in that overfitting to the training typically happens only if very complicated models are learned, or functions with discontinuities are used (e.g., protected division [14]). We adopt a typical 75%-25% random splitting of the examples into training and test set for a given run.

Each experiment consists of 30 independent runs. To assess if the results of one experiment are significantly better or worse than the ones of another, we use the non-parametric Wilcoxon signed-rank test [6], pairing runs by random seed. The random seed determines the train-test split and the sampling of the initial population. We say a result is significant if the $p$-value of the statistical test is below a threshold $\tau$. We use $\tau = 0.05$, and further apply the Bonferroni correction method, to prevent false positive claims [6].

We run the experiments on a machine with two Intel® Xeon® CPU E5-2699 v4 @ 2.20GHz, and 630 GB of RAM. Big amounts of memory are needed to use the intermediate output caching, as single runs can already employ a few GB of memory.

[1]https://archive.ics.uci.edu/ml/index.php

**Table 1: Parameter settings of GP.**

| Parameter | Setting |
|---|---|
| # Generations / time limit | 100 / 1000 s |
| Population size | 500 |
| Function set | $\{+, -, \times, \div_{\text{AQ}}\}$ |
| Terminal set | Features ∪ ERC |
| ERC sample method | $\mathbb{U}$[min(Features), max(Features)] |
| Initialization | Ramped Half-Half 2–6 |
| Maximum tree height | 12 |
| Maximum number of nodes | 500 |
| Selection | Tournament 4 & Elitism 1 |
| Variation | RDO with rate 1.0 |
| Intermediate output caching | Active |

**Table 2: Functions considered and their inversions.**

| Function | Direct | Inversion(s) |
|---|---|---|
| $+$ | $a_i + a_j = o$ | $a_i = o - a_j$ |
| $-$ | $a_i - a_j = o$ | $a_i = o + a_j$, $a_j = a_i - o$ |
| $\times$ | $a_i \times a_j = o$ | $a_i = o/a_j$ if $o$, $a_j \neq 0$ |
| | | $*$ if $o$, $a_j = 0$ |
| | | impossible if $o \neq 0$, $a_j = 0$ |
| $\div_{\text{AQ}}$ | $a_i / \sqrt{1 + a_j^2} = o$ | $a_i = o \times \sqrt{1 + a_j^2}$ |
| | | $a_j = +\sqrt{(a_i/o)^2 - 1}$ if $o \neq 0$, $(a_i/o)^2 \geq 1$ |
| | | impossible if $o = 0$ or $(a_i/o)^2 < 1$ |

**Table 3: Real-world benchmark datasets.**

| (Abbreviation) Name | Examples ($n$) | Features | $Var(\mathbf{y})$ | Link |
|---|---|---|---|---|
| (A) Airfoil | 1503 | 5 | $4.756 \cdot 10$ | goo.gl/uNMLv3 |
| (B) Boston housing | 506 | 13 | $8.442 \cdot 10$ | goo.gl/KxCnq1 |
| (C) Concrete strength | 1030 | 8 | $2.788 \cdot 10^2$ | goo.gl/Gjq9oN |
| (D) Dow chemical | 1066 | 57 | $1.228 \cdot 10^{-1}$ | goo.gl/9D2z3b |
| (Ec) Energy cooling | 768 | 8 | $9.039 \cdot 10$ | goo.gl/ANV6dW |
| (Eh) Energy heating | 768 | 8 | $1.017 \cdot 10^2$ | goo.gl/ANV6dW |
| (T) Tower | 4999 | 26 | $6.518 \cdot 10^{-1}$ | goo.gl/9D2z3b |
| (Wr) Wine red | 1599 | 11 | $7.842 \cdot 10^{-1}$ | goo.gl/inDsCE |
| (Ww) Wine white | 4899 | 11 | $7.702 \cdot 10^3$ | goo.gl/inDsCE |
| (Y) Yacht hydrodynamics | 308 | 6 | $2.291 \cdot 10^2$ | goo.gl/cmkRor |

## 7 RESULTS

We proceed by showing the results of the following experiments. Firstly we consider whether using LS in synergy with SB is beneficial compared to using it independently. Secondly, we compare all configurations of SB-based GP with SGP, by fixing the maximum number of generations, and observing the time taken. Thirdly, we repeat the previous experiment, but this time using a fixed time budget, to take into account computational expensiveness.

### 7.1 Independent vs synergetic linear scaling with semantic backpropagation

Table 4 shows the median error obtained by end-of-run best trees found using SB-based GP without LS (noLS), with LS but independently from SB (iLS), and with LS in synergy with SB (sLS),

**Table 4: Training and test median NMSEs for SB-based GP without LS (noLS), with LS used independently (iLS), and with LS used in synergy with SB (sLS). Underlined results are best in that no other is significantly better.**

| | Train NMSE | | | Test NMSE | | |
|---|---|---|---|---|---|---|
| | noLS | iLS | sLS | noLS | iLS | sLS |
| A | 41 | 29 | 22 | 43 | 32 | 29 |
| B | 27 | 17 | 14 | 27 | 21 | 16 |
| C | 34 | 19 | 15 | 37 | 21 | 18 |
| D | 71 | 29 | 20 | 69 | 28 | 21 |
| Ec | 9.9 | 6.9 | 4.9 | 8.7 | 7.1 | 5.4 |
| Eh | 6.8 | 4.0 | 5.4 | 8.7 | 6.2 | 7.1 |
| T | 16 | 14 | 13 | 17 | 14 | 14 |
| Wr | 69 | 63 | 60 | 65 | 63 | 62 |
| Ww | 75 | 69 | 67 | 78 | 71 | 70 |
| Y | .62 | .44 | .40 | .94 | .62 | .61 |
| # Best | 0 | 2 | 10 | 0 | 2 | 10 |

after 100 generations. The results are reported in terms of variance-Normalized MSE (NMSE), given by dividing the MSE by the variance of the dependent variable $\mathbf{y}$, to have results of similar order of magnitude, and multiplying by 100.

Evidently, iLS has much better training and test performance compared to noLS. This is always significant w.r.t. training NMSE, and is also significant on all datasets but for Boston at test time. However, to use LS in synergy with SB is even better, significantly outperforming both noLS (all cases) and iLS on 8/10 datasets both at training and test time. Our hypothesis that using LS in synergy with SB is beneficial is therefore experimentally confirmed.

## 7.2 SB-based GP vs standard GP

The next results present the outcome of comparing SB-based GP with SGP, with and without using LS to scale the error and within library search. We first consider a limit of 100 generations.

*7.2.1 Budget of 100 generations.* Figure 4 shows, for each dataset, the evolution of the best training fitness for SGP, SGP with LS ($\text{SGP}_{+\text{LS}}$), SB-based GP with traditional RDO (RDO), RDO using LS in synergy during backpropagation ($\text{RDO}_{+\text{LS}}$), RDO using LS within library search ($\text{RDO}^{\text{xLS}}$), and RDO using both LS in synergy with backpropagation and within library search ($\text{RDO}^{\text{xLS}}_{+\text{LS}}$).

RDO and SGP are complementary: one is better than the other on half datasets. However, on Tower and Yacht, SGP has much larger errors. In some cases (Airfoil, Boston, Concrete, Wine white, Yacht), it is noticeable that the error of SGP levels off less markedly than the one of RDO, thus a larger generational budget may favor SGP. $\text{RDO}_{+\text{LS}}$ is better than $\text{SGP}_{+\text{LS}}$ on all datasets but Yacht.

$\text{RDO}^{\text{xLS}}$ and $\text{RDO}^{\text{xLS}}_{+\text{LS}}$ are consistently the best performing, with the second reaching slightly smaller errors than the first. Moreover, both algorithms have smaller variances than $\text{RDO}_{(+\text{LS})}$. This is because the use of LS within library search (xLS) dynamically improves the expressiveness of the library for any desired output that is searched. Without xLS, the expressiveness of the library is more aleatory, as it only depends on the subtrees from the population.

Table 5 shows training and test NMSEs of end-of-run best trees. The training errors reflect what already seen in Fig. 4. Test errors are typically similar to training ones, for all the algorithms. $\text{RDO}^{\text{xLS}}_{+\text{LS}}$ is the best performing, while $\text{RDO}^{\text{xLS}}$ is the second best. On Wine red at test time, $\text{SGP}_{+\text{LS}}$ is preferable over $\text{RDO}^{\text{xLS}}_{+\text{LS}}$, which indicates that the latter overfits (slightly).

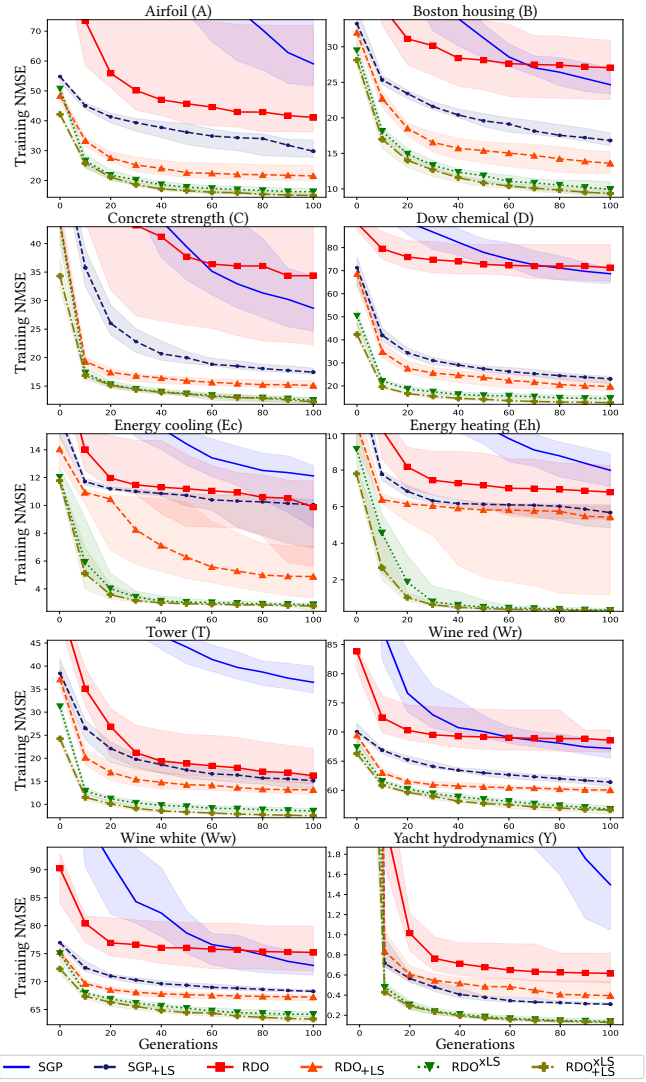


**Figure 4: Median best training NMSE (25th and 75th percentiles within shaded area) in 100 generations.**

*7.2.2 Time taken by 100 generations.* Figure 5 show the time taken to perform 100 generations for the algorithms. The difference between the times taken by SGP and $\text{SGP}_{+\text{LS}}$ and the various configurations of RDO is very large. For Yacht, that has 308 examples, RDO takes around 20 times longer than $\text{SGP}_{(+\text{LS})}$; For Tower, that has 4999 examples, RDO takes around 100 times longer than $\text{SGP}_{(+\text{LS})}$. This result strongly motivates the need for a time-based comparison between SGP and RDO configurations, for fairness.

The use of LS in addition to RDO, or within library search, does, on average, increase running times. However, these running times are not too dissimilar if put in perspective to the times taken by $\text{SGP}_{(+\text{LS})}$. This is expected because +LS and xLS do not affect computational time bounds. RDO and $\text{RDO}_{+\text{LS}}$ have larger variations (some of the extreme time points of RDO are considered outliers).

**Table 5: Training and test median NMSEs, for the experiments with a budget of 100 generations. Underlined results are best in that no other is significantly better.**

| | Train NMSE | | | | | | Test NMSE | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | SGP | $SGP_{+LS}$ | RDO | $RDO_{+LS}$ | $RDO^{xLS}$ | $RDO^{xLS}_{+LS}$ | SGP | $SGP_{+LS}$ | RDO | $RDO_{+LS}$ | $RDO^{xLS}$ | $RDO^{xLS}_{+LS}$ |
| A | 59 | 30 | 41 | 22 | 16 | 15 | 60 | 32 | 43 | 29 | 20 | 19 |
| B | 25 | 17 | 27 | 14 | 10 | 9.4 | 23 | 17 | 27 | 16 | 16 | 14 |
| C | 29 | 17 | 34 | 15 | 13 | 12 | 30 | 20 | 37 | 18 | 15 | 15 |
| D | 69 | 23 | 71 | 20 | 15 | 13 | 65 | 23 | 69 | 21 | 16 | 15 |
| Ec | 12 | 10 | 9.9 | 4.9 | 2.9 | 2.8 | 11 | 9.0 | 8.7 | 5.4 | 3.4 | 3.2 |
| Eh | 8.0 | 5.7 | 6.8 | 5.4 | .35 | .32 | 10 | 7.6 | 8.7 | 7.1 | .50 | .40 |
| T | 36 | 15 | 16 | 13 | 8.6 | 7.5 | 36 | 15 | 17 | 14 | 10 | 9.5 |
| Wr | 67 | 61 | 69 | 60 | 57 | 57 | 65 | 62 | 65 | 62 | 62 | 65 |
| Ww | 73 | 68 | 75 | 67 | 64 | 63 | 75 | 70 | 78 | 70 | 69 | 68 |
| Y | 1.5 | .31 | .62 | .40 | .14 | .13 | 1.9 | .40 | .90 | .60 | .30 | .30 |
| # Best | 0 | 0 | 0 | 0 | 1 | 10 | 0 | 1 | 0 | 1 | 4 | 8 |

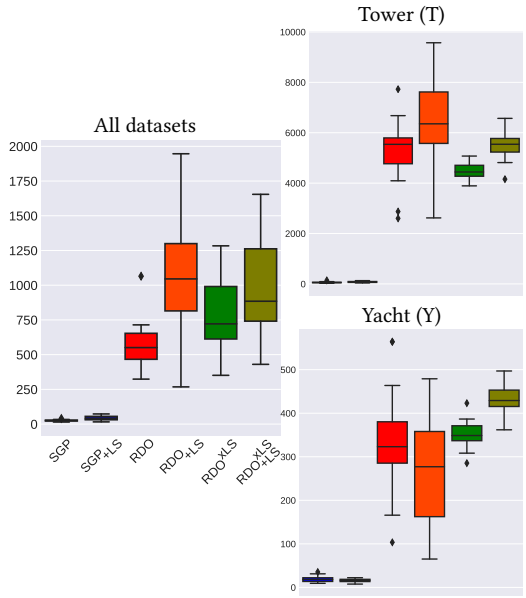


**Figure 5: Time (seconds) to complete 100 generations. Left: Mean time over all datasets; Right: Time by the 30 runs on Tower (top), and Yacht (bottom); Boxes extend from the 25th to the 75th percentiles (inner bar is the 50th), whiskers from the 10th to the 90th. Diamonds are outliers.**

These variations in time are linked to the variations already seen in terms of fitness (e.g., see the Energy datasets in Fig. 4).

*7.2.3 Budget of 1000 seconds.* The evolution of the best training fitness in time is reported in Figure 6, for each dataset and algorithm. The conclusions that can be drawn from these results are different from the ones based on a generational limit. For SGP, the use of a time limit of 1000 seconds seems more appropriate than the limit of 100 generations, since the fitness tends to plateau more in this case (this is particularly evident for the smallest dataset Yacht).

Now, RDO performs markedly worse than SGP, and $RDO_{+LS}$ is also worse than $SGP_{+LS}$, with the latter typically achieving close performance to $RDO^{xLS}$. While in a time-based comparison RDO performs quite poorly, it is interesting to see that, instead, $RDO^{xLS}$ and $RDO^{xLS}_{+LS}$ still perform very well. Indeed, the inclusion of LS within library search makes variation so effective that, even if library search itself becomes slower, fitter trees are discovered sooner. While it is perhaps not surprising that xLS makes variation improve, it is interesting to see the extent of this improvement.

**Table 6: Training and test median NMSEs, for the experiments with a budget of 1000 seconds. Underlined results are best in that no other is significantly better.**

| | Train NMSE | | | | | | Test NMSE | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | SGP | $SGP_{+LS}$ | RDO | $RDO_{+LS}$ | $RDO^{xLS}$ | $RDO^{xLS}_{+LS}$ | SGP | $SGP_{+LS}$ | RDO | $RDO_{+LS}$ | $RDO^{xLS}$ | $RDO^{xLS}_{+LS}$ |
| A | 27 | 19 | 41 | 23 | 17 | 16 | 32 | 22 | 44 | 30 | 20 | 20 |
| B | 13 | 9.0 | 25 | 13 | 10 | 8.7 | 17 | 14 | 26 | 15 | 16 | 14 |
| C | 16 | 13 | 30 | 15 | 12 | 12 | 18 | 16 | 37 | 19 | 15 | 15 |
| D | 38 | 14 | 71 | 20 | 15 | 13 | 43 | 16 | 68 | 22 | 16 | 15 |
| Ec | 3.8 | 3.3 | 8.5 | 4.7 | 2.8 | 2.6 | 4.8 | 3.8 | 7.4 | 5.6 | 3.4 | 3.2 |
| Eh | 1.2 | .77 | 6.5 | 4.1 | .33 | .28 | 1.5 | .91 | 8.4 | 5.4 | .45 | .35 |
| T | 22 | 11 | 23 | 18 | 10 | 9.0 | 22 | 12 | 23 | 19 | 12 | 11 |
| Wr | 60 | 58 | 69 | 60 | 57 | 57 | 63 | 62 | 65 | 62 | 62 | 63 |
| Ww | 68 | 66 | 76 | 68 | 66 | 66 | 70 | 69 | 79 | 70 | 69 | 69 |
| Y | .27 | .16 | .50 | .35 | .12 | .10 | .49 | .33 | .80 | .55 | .35 | .33 |
| # Best | 0 | 0 | 0 | 0 | 2 | 10 | 0 | 3 | 0 | 2 | 4 | 8 |

Table 6 summarizes both training and test NMSEs of end-of-run best trees. The tests for statistical significance confirm what already seen in the training fitness convergence plots of Fig. 6: $RDO^{xLS}_{+LS}$ is the top performing algorithm, followed by $RDO^{xLS}$. In terms of error magnitudes, $SGP_{+LS}$ is relatively close to $RDO^{xLS}$ and $RDO^{xLS}_{+LS}$ (yet often significantly worse), compared to SGP and RDO w/o LS.

When it comes to generalization, $RDO^{xLS}_{+LS}$ is still preferable, as it is significantly worse than another algorithm only on 2 datasets, by relatively small magnitudes. $SGP_{+LS}$ leads to very good generalization on 3 out of 10 datasets, indicating that $RDO^{xLS}_{+LS}$, which was better at training time, delivered slightly overfitted trees.

All in all, our results show that scaling the trees during library search is extremely valuable for RDO. In addition, to consider LS when backpropagating, i.e., $RDO^{xLS}_{+LS}$, gives a further edge.

## 8 DISCUSSION

We found that a comparison between RDO and SGP on real-world datasets strongly depends on how this comparison is framed. With a typical budget of 100 generations, the algorithms perform complementarily. Instead, when the comparison is framed in terms of time, RDO performs worse than SGP. Our proposal of incorporating LS within the mechanisms of RDO makes the algorithm much more effective even if extra computations take place, and makes it capable of outperforming all the other algorithms.

We now discuss some limitations of this paper. To begin, we used typical settings for the parameters of GP. One may wonder whether our findings do generalize to other configurations. Population sizing is perhaps the most interesting aspect to consider [10], especially when using a population-based library (which is composed of all subtrees with unique output from the population). If a library is large enough, i.e., if it has enough representative power, the adoption of LS may become redundant. However, because LS

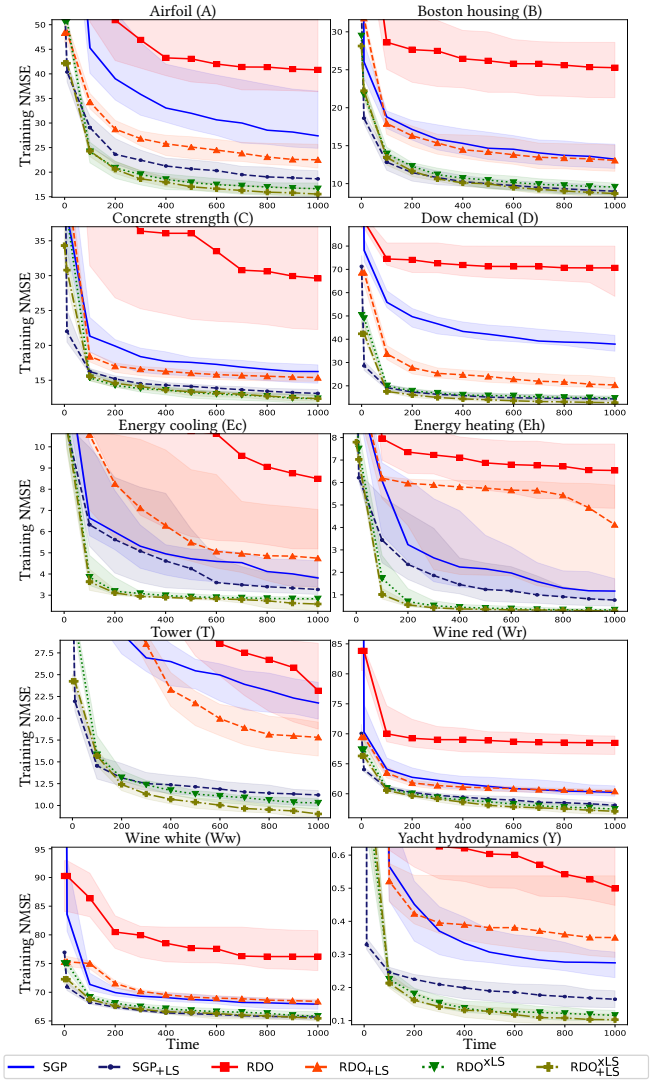


**Figure 6: Median best training NMSE (25th and 75th percentiles within shaded area) in 1000 seconds.**

applies a linear transformation that is *optimal*, we argue that populations and libraries will likely need to grow too big to be practically usable to compete with LS. As to other parameters, we believe that the magnitude of our results, as well as the small variances found along the runs, strongly indicate that the use of LS within library search will remain beneficial for many other parameter settings.

Another limit of our approach, in particular of using LS within library search, is the growth of tree size. Any time a tree is retrieved from the library, four nodes are added to incorporate the effect of LS. Larger trees are slower to evaluate (and require more memory to cache intermediate outputs), and are also less likely to lead to interpretable expressions. Interpretability of machine learning models can be a relevant aspect for practical applications, e.g., in healthcare [16, 23]. By including LS within library search, we did find trees to grow bigger, with the best trees found by $\text{RDO}^{\text{xLS}}_{\text{+LS}}$ being on average 1.1 times larger than the the ones found by $\text{SGP}_{\text{+LS}}$ in the time-based comparison. We claim that this difference in size is largely unimportant. Both algorithms deliver quite big trees anyway, with approximately 325 nodes for $\text{SGP}_{\text{+LS}}$ and 360 nodes for $\text{RDO}^{\text{xLS}}_{\text{+LS}}$, on average. From a performance perspective, the increment in time taken to evaluate a larger tree is limited, but may become noticeable for much larger datasets than the ones we considered. As to interpretability, the algorithms deliver trees that are equivalently too large to result in interpretable expressions.

Future work may therefore focus on reducing tree size, e.g., by exploring the inclusion of bloat control methods [17], or by expressing preference for smaller trees as a secondary objective [7]. However, if having trees with only a few dozen nodes is truly desired, we believe substantially different approaches to GP may need to be taken, such as modern model-based GP [23, 24].

Another aspect worth investigating is the use of more efficient data structures to implement the library. In [15], $k$-d trees are used [2, 9]. We did experiments with this data structure, and although searching $k$-d trees may not be quick for datasets with many examples [9], we did observe speed ups for the datasets we considered. Unfortunately, LS cannot be used within $k$-d tree search. This is because a $k$-d tree is built exploiting the fixed distribution of tree outputs, to cut exploration branches when searching. To apply LS means to dynamically change such a distribution.

We did experiments with adopting $k$-d trees jointly with the computation of *only* the optimal translation coefficient $a$. This can be achieved by (i) Subtracting the mean of the output $\mathbf{o}$ of each library tree prior to building the $k$-d tree; (ii) Subtracting the mean of the desired output $\mathbf{d}$ prior to $k$-d tree search; (iii) Incorporating the addition of $a = \bar{\mathbf{d}} - \bar{\mathbf{o}}$ to the tree returned by the search. This achieves the optimal translation (see Eq 2). However, we found this to be less effective than using LS (which also computes $b$) within traditional library search. To find an efficient data structure that enables the use of LS or of a similarly powerful method, as well as investigating code parallelization, may allow to use SB-based GP for large scale symbolic regression.

## 9 CONCLUSION

We presented the use of Linear Scaling (LS) in synergy with Semantic Backpropagation (SB), and within library search, in Genetic Programming (GP) for symbolic regression. We validated the proposed adaptations on ten real-world datasets, comparing various GP configurations using a generational and a time budget. We found that incorporating LS within SB-based GP leads to much lower errors in both cases, and outperforms the use of traditional variation operators. Lastly, the cost incurred by our adaptations is limited, as the asymptotic time bounds of SB-based GP remain unchanged.

## ACKNOWLEDGMENTS

Financial support for this work was provided by Stichting Kinderen Kankervrij (Children Cancer-free Foundation), project #187. We further thank the Maurits en Anna de Kock Foundation for financing a high-performance computing system, and SURFsara for the support in using the Lisa Compute Cluster. We also thank professor Krzysztof Krawiec from the Poznan University of Technology, Poland, for the insightful exchanges on semantic backpropagation-based genetic programming.

## REFERENCES


[1] Francesco Archetti, Ilaria Giordani, and Leonardo Vanneschi. 2010. Genetic programming for anticancer therapeutic response prediction using the NCI-60 dataset. *Computers & Operations Research* 37, 8 (2010), 1395–1405.

[2] Jon Louis Bentley. 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM* 18, 9 (1975), 509–517.

[3] Qi Chen, Bing Xue, and Mengjie Zhang. 2018. Improving Generalisation of Genetic Programming for Symbolic Regression with Angle-Driven Geometric Semantic Operators. *IEEE Transactions on Evolutionary Computation* (2018).

[4] Qi Chen, Mengjie Zhang, and Bing Xue. 2017. Geometric Semantic Genetic Programming with Perpendicular Crossover and Random Segment Mutation for Symbolic Regression. In *Asia-Pacific Conference on Simulated Evolution and Learning*. Springer, 422–434.

[5] Dan Costelloe and Conor Ryan. 2009. On improving generalisation in genetic programming. In *European Conference on Genetic Programming*. Springer, 61–72.

[6] Janez Demšar. 2006. Statistical comparisons of classifiers over multiple data sets. *Journal of Machine learning research* 7, Jan (2006), 1–30.

[7] Anikó Ekárt and Sandor Z. Nemeth. 2001. Selection based on the pareto nondomination criterion for controlling code growth in genetic programming. *Genetic Programming and Evolvable Machines* 2, 1 (2001), 61–73.

[8] Robyn Ffrancon and Marc Schoenauer. 2015. Memetic semantic genetic programming. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*. ACM, 1023–1030.

[9] Jerome H Friedman, Jon Louis Bentley, and Raphael Ari Finkel. 1976. An algorithm for finding best matches in logarithmic time. *ACM Trans. Math. Software* 3, SLAC-PUB-1549-REV. 2 (1976), 209–226.

[10] George Harik, Erick Cantú-Paz, David E Goldberg, and Brad L Miller. 1999. The gambler's ruin problem, genetic algorithms, and the sizing of populations. *Evolutionary Computation* 7, 3 (1999), 231–253.

[11] Quang Nhat Huynh, Shelvin Chand, Hemant Kumar Singh, and Tapabrata Ray. 2018. Genetic Programming with Mixed Integer Linear Programming Based Library Search. *IEEE Transactions on Evolutionary Computation* (2018).

[12] Maarten Keijzer. 2003. Improving symbolic regression with interval arithmetic and linear scaling. *Genetic Programming* (2003), 275–299.

[13] Maarten Keijzer. 2004. Scaled symbolic regression. *Genetic Programming and Evolvable Machines* 5, 3 (2004), 259–269.

[14] John R. Koza. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA.

[15] Krzysztof Krawiec and Tomasz Pawlak. 2013. Approximating geometric crossover by semantic backpropagation. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*. ACM, 941–948.

[16] Zachary C Lipton. 2016. The mythos of model interpretability. *arXiv preprint arXiv:1606.03490* (2016).

[17] Sean Luke and Liviu Panait. 2006. A comparison of bloat control methods for genetic programming. *Evolutionary Computation* 14, 3 (2006), 309–344.

[18] Ji Ni, Russ H Drieberg, Peter Rockett, et al. 2013. The Use of an Analytic Quotient Operator in Genetic Programming. *IEEE Transactions on Evolutionary Computation* 17, 1 (2013), 146–152.

[19] Tomasz P. Pawlak, Bartosz Wieloch, and Krzysztof Krawiec. 2015. Semantic backpropagation for designing search operators in genetic programming. *IEEE Transactions on Evolutionary Computation* 19, 3 (2015), 326–340.

[20] Riccardo Poli, William B Langdon, Nicholas F McPhee, and John R Koza. 2008. *A field guide to genetic programming*. Lulu. com.

[21] Adil Raja, R Muhammad Atif Azad, Colin Flanagan, and Conor Ryan. 2007. Real-time, non-intrusive evaluation of VoIP. In *European Conference on Genetic Programming*. Springer, 217–228.

[22] Marcin Szubert, Anuradha Kodali, Sangram Ganguly, Kamalika Das, and Josh C Bongard. 2016. Semantic Forward Propagation for Symbolic Regression. In *International Conference on Parallel Problem Solving from Nature*. Springer, 364–374.

[23] Marco Virgolin, Tanja Alderliesten, Arjan Bel, Cees Witteveen, and Peter A.N. Bosman. 2018. Symbolic regression and feature construction with GP-GOMEA applied to radiotherapy dose reconstruction of childhood cancer survivors. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '18)*. ACM, 1395–1402.

[24] Marco Virgolin, Tanja Alderliesten, Cees Witteveen, and Peter A.N. 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 (GECCO '17)*. ACM, New York, NY, USA, 1041–1048.

[25] David R White, James Mcdermott, Mauro Castelli, Luca Manzoni, Brian W Goldman, Gabriel Kronberger, Wojciech Jaśkowski, Una-May O'Reilly, and Sean Luke. 2013. Better GP benchmarks: community survey results and proposals. *Genetic Programming and Evolvable Machines* 14, 1 (2013), 3–29.

[26] Bartosz Wieloch and Krzysztof Krawiec. 2013. Running programs backwards: instruction inversion for effective search in semantic spaces. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '15)*. ACM, 1013–1020.