



Genetic programming is naturally suited to evolve bagging ensembles

Downloaded from: <https://research.chalmers.se>, 2026-04-12 00:58 UTC

Citation for the original published paper (version of record):

Virgolin, M. (2021). Genetic programming is naturally suited to evolve bagging ensembles. GECCO 2021 - Proceedings of the 2021 Genetic and Evolutionary Computation Conference: 830-839.
<http://dx.doi.org/10.1145/3449639.3459278>

N.B. When citing this work, cite the original published paper.

Genetic Programming is Naturally Suited to Evolve Bagging Ensembles

Marco Virgolin
marco.virgolin@chalmers.se
Chalmers University of Technology
Gothenburg, Sweden

ABSTRACT

Learning ensembles by bagging can substantially improve the generalization performance of low-bias, high-variance estimators, including those evolved by Genetic Programming (GP). To be efficient, modern GP algorithms for evolving (bagging) ensembles typically rely on several (often inter-connected) mechanisms and respective hyper-parameters, ultimately compromising ease of use. In this paper, we provide experimental evidence that such complexity might not be warranted. We show that minor changes to fitness evaluation and selection are sufficient to make a simple and otherwise-traditional GP algorithm evolve ensembles efficiently. The key to our proposal is to exploit the way bagging works to compute, for each individual in the population, multiple fitness values (instead of one) at a cost that is only marginally higher than the one of a normal fitness evaluation. Experimental comparisons on classification and regression tasks taken and reproduced from prior studies show that our algorithm fares very well against state-of-the-art ensemble and non-ensemble GP algorithms. We further provide insights into the proposed approach by (i) scaling the ensemble size, (ii) ablating the changes to selection, (iii) observing the evolvability induced by traditional subtree variation.

Code: <https://github.com/marcovirgolin/2SEGP>.

CCS CONCEPTS

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

KEYWORDS

Genetic programming, ensemble learning, machine learning, bagging, evolutionary algorithms

ACM Reference Format:

Marco Virgolin. 2021. Genetic Programming is Naturally Suited to Evolve Bagging Ensembles. In *2021 Genetic and Evolutionary Computation Conference (GECCO '21)*, July 10–14, 2021, Lille, France. ACM, New York, NY, USA, 10 pages. <https://doi.org/10.1145/3449639.3459278>

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 '21, July 10–14, 2021, Lille, France

© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8350-9/21/07...\$15.00
<https://doi.org/10.1145/3449639.3459278>

1 INTRODUCTION

Learning ensembles by *bagging*, i.e., aggregating the predictions of low-bias, high-variance estimators fitted on different samples of the training set (see Sec. 3.1 for a more detailed description), can improve generalization performance significantly [6, 7, 19, 50]. Random forests are a remarkable example of this [5, 8]. At the same time, mixed results have been found when using deep neural networks [13, 23, 41, 62]. For Genetic Programming (GP) [36, 53], bagging has generally been found to be beneficial [17, 28, 34, 56]. Since in a *classic* GP algorithm the outcome of the evolution is *one* best-found individual (i.e., the estimator that best fits the training set), perhaps the simplest way to build an ensemble of GP individuals is to evolve multiple populations independently, and aggregate the outcomes. However, since a GP population can naturally host diverse individuals, it makes sense to seek ways to evolve the ensemble in one go and save substantial computational resources.

Many ensemble learning GP-based approaches have been proposed so far (see Sec. 2 for an overview). We can broadly categorize them in two classes: *Simple Independent Ensemble Learning Approaches* (SIEL-Apps), and *Complex Simultaneous Ensemble Learning Algorithms* (CSEL-Algs). SIEL-Apps form an ensemble of estimators by repeating the execution of a (typically classic) GP algorithm that produces, each time, a single estimator. As said before, this idea is simple but inefficient. Instead, CSEL-Algs make use of a number of novel mechanisms and respective hyper-parameters to obtain an ensemble in one go. For this reason, CSEL-Algs can be very efficient, but also quite complex and thus difficult to adopt in practical applications. Moreover, from a scientific standpoint, it may be hard to assess which moving-parts of a CSEL-Alg are really needed and which are not.

In this paper, we seek to obtain the best of both worlds: A GP algorithm that learns ensembles efficiently (e.g., without repeating multiple evolutions) and is simple enough to be thought as a possible minimal/natural extension of classic GP. Specifically, given a classic tree-based GP algorithm, we introduce only (arguably) minor modifications to fitness evaluation and selection, with the goal of **making the population specialize uniformly across different realizations of the training set** (in the context of bagging, these are called *bootstrap samples*, see Sec. 3.1). The proposed modifications are time-efficient. We call the resulting algorithm *Simple Simultaneous Ensemble Genetic Programming* (2SEGP) and show that, despite its simplicity, 2SEGP is competitive with State-of-the-Art (SotA) ensemble and non-ensemble GP algorithms. We do this by reporting and reproducing results from recent literature on real-world benchmark classification and regression datasets. Moreover, to better understand what matters when learning bagging ensembles in GP, we include experiments that dissect our algorithm.

2 RELATED WORK

In this paper we focus on ensemble learning intended as bagging (see Sec. 3.1), when GP is used to evolve the estimators. We do not consider ensemble learning intended as boosting, i.e., the iterative fitting of weak estimators to (weighted) residuals [14, 24, 25]. For the reader interested in boosting GP, we refer, e.g., to [15, 21, 28, 51, 57]. Similarly, we do not consider works where, even if GP was used to decide how to aggregate the estimators, these were learned by other algorithms than GP [3, 4, 31, 43, 44].

Starting with SIEL-Apps, we remark that the works in this category mostly focus on *how the aggregation of GP-evolved estimators can be improved*, rather than on how to evolve ensembles efficiently. For example, some early works look into improving the ensemble prediction quality by weighing member predictions by a measure of confidence [28] or by bypassing outlier member predictions [34]. Further investigations have been carried out across problems of different nature, in [29, 30, 60]. An SIEL-App is also used in [66], yet this time with a non-classic GP algorithm where individuals are linear combinations of multiple trees and the evolution is made scalable by leveraging on-line, parallel computing. Other works in this category are [35, 64, 77], respectively for hybridization with multi-objective evolution, incomplete data, and large-scale data.

CSEL-Algs are of most interest w.r.t. the present work as they attempt to evolve an ensemble in an efficient manner. In [2], e.g., multi-objective GP is used to build ensembles where the members are Pareto non-dominated individuals. Importantly, having multiple objectives is a prerequisite for this proposal (not the case here). Multifactorial GP is used in [74] to evolve ensembles of decision tree-like individuals that each interpret the dataset features differently. More recently, [73] proposed the Diverse Niching Genetic Programming (DivNichGP) algorithm, which works in single-objective and manages to obtain an ensemble by maintaining population diversity by (i) Using bootstrap sampling every generation to constantly vary the training data distribution, and (ii) Including a niching mechanism. Niching is further used at termination in order to pick the final ensemble members from the population, and requires two dedicated hyper-parameters to be set. Another recent investigation is [17], where ensembles are learned to reduce the typical susceptibility of symbolic regression GP algorithms to outliers. In that work, spatially-clustered individuals (e.g., as neighboring nodes of a toroidal graph) compete in fitting different bootstrap samples [63]. This algorithm requires to choose the graph and cluster structure as well as the way computational resources should be distributed on the graph nodes. Lastly, in [56] ensemble learning is realized by the simultaneous co-evolution of a population of estimators (trees), and a population of ensemble candidates (forests). For this algorithm, alongside the hyper-parameters for the population of trees, one needs to set the hyper-parameters for the population of forests (e.g., for variation, selection, and voting method).

We remark that, in order to ameliorate for the complexity introduced in CSEL-Algs, the respective works provide recommendations on default hyper-parameter settings. Even so, we believe that these algorithms can still be considered sufficiently complex that pursuing a simpler approach remains a worthwhile endeavour. We include the three CSEL-Algs from [17, 56, 73] (among other GP algorithms) in our comparisons.

3 LEARNING BAGGING ENSEMBLES BY MINOR MODIFICATIONS TO CLASSIC GP

We now describe how, taken a classic GP algorithm that returns a single best-found estimator, one can evolve bagging ensembles. In other words, how to obtain 2SEGP from classic GP. We assume the reader to be familiar with the workings of a classic tree-based GP algorithm, and refer, e.g., to [53] (Chapters 2–4).

The backbone of our proposal consists of two aspects: (i) *Evaluate a same individual according to different realizations of the training set (i.e., bootstrap samples)*; and (ii) *Let the population improve uniformly across these realizations*. To achieve these aspects, we only modify fitness evaluation (we also describe the use of linear scaling [32] as it is very useful in practice) and selection. We do not make any changes to variation: Any parent can mate with any other (using classic subtree crossover), and any type of genetic material can be used to mutate any individual (using classic subtree mutation). Our intuition is that exchanging genetic material between estimators that are best on different samples of the training set is not detrimental because these samples are themselves similar to one another (we provide insights about this in Sec. 7.3).

We proceed by recalling how bagging works, followed by describing the modifications we propose, for the sake of clarity, first to selection and then to fitness evaluation.

3.1 Bagging

As aforementioned, we focus on learning ensembles by bagging, i.e., bootstrap aggregating [7]. We use traditional bootstrap, i.e., we obtain β realizations of the training set $\mathbb{T}_1, \dots, \mathbb{T}_\beta$, each with as many observations as the original training set $\mathbb{T} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$, by uniformly sampling from \mathbb{T} with replacement. Aggregation of predictions is performed the traditional way, i.e., by majority voting (i.e., mode) for classification, and averaging for regression. One run of our algorithm will produce an ensemble of β members where each member is the best-found individual (i.e., elite) according to the fitness measured on the bootstrap sample \mathbb{T}_j , with $j = 1, \dots, \beta$.

3.2 Selection for uniform progress across the bootstrap samples

We employ a remarkably simple modification of truncation selection that is applied after the offspring population has been obtained by variation of the parent population, i.e., in a $(\mu + \lambda)$ fashion. The main idea is to select individuals in such a way that progress is uniform across all the bootstrap samples $\mathbb{T}_1, \dots, \mathbb{T}_\beta$. To this end, we now make the assumption that each individual does not have a single fitness value, rather, it has β of them, one per bootstrap sample \mathbb{T}_j . We show how these β fitness values can be computed efficiently in Sec. 3.3.

Pseudocode for the modified truncation selection is given in Algorithm 1. Very simply, we perform β truncation selections, each focused on one of the β fitness values, and where the (n_{pop}/β) top-ranking individuals are chosen each time. Note that this selection ensures weakly monotonic fitness decrease across all the bootstrap samples. Note also that a same individual can obtain multiple copies if it has fitness values such that it is top-ranking according to multiple bootstrap samples.

Lastly, one can see that the computational complexity of this selection method is determined by sorting the population β times and copying individuals, i.e., $O(\beta n_{\text{pop}} \log n_{\text{pop}} + n_{\text{pop}} \ell)$, under the assumption that ℓ is the (worst-case) size of an individual (in the case of tree-based GP, the number of nodes). As we will show in Sec. 3.3 below, the cost of fitness evaluation over the entire population will dominate the cost of selection. See Sec. 7.2 for ablations.

Algorithm 1: Our simple extension of truncation selection.

input : \mathbb{P} (parent pop.), \mathbb{O} (offspring pop.), β (ensemble size)
output: \mathbb{P}' (new pop. of selected individuals)

```

1  $\mathbb{Q} = \text{join}(\mathbb{P}, \mathbb{O});$ 
2  $\mathbb{P}' \leftarrow [];$ 
3 for  $j \in 1, 2, \dots, \beta$  do
4   sort  $\mathbb{Q}$  according to the  $j$ th fitness value;
5   for  $k \in 1, 2, \dots, (n_{\text{pop}}/\beta)$  do
6      $\mathbb{P}' \leftarrow \text{join}(\mathbb{P}', [\mathbb{Q}_k]);$ 
7 return  $\mathbb{P}';$ 

```

3.3 Fitness evaluation on all bootstrap samples

A typical fitness evaluation in GP comprises (i) Computing the output of the individual in consideration; (ii) Computing the loss function between the output and the label. Both steps are performed using the original training set $\mathbb{T} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$. Recall that the computation cost of step (i) is $O(\ell n)$, because we need to compute the ℓ operations that compose the individual for each observation in the training set. Step (ii) takes $O(n)$ but is additive, thus the total asymptotic cost ultimately amounts to $O(\ell n)$.

Since we wish the population to evolve uniformly well across the bootstrap samples, our selection method needs each individual to have a fitness value for each bootstrap sample. In other words, we need to compute the fitness w.r.t. $\mathbb{T}_1, \dots, \mathbb{T}_\beta$. A naive solution would be to repeat steps (i) and (ii) for each \mathbb{T}_j , leading to a time cost of $O(\beta \ell n)$; Ultimately the same cost an SIEL-App would have (although distributed across multiple evolutions).

To improve upon the naive cost $O(\beta \ell n)$, we make the following observation. In many machine learning algorithms, the specific realization of the training set determines the structure of the estimator that will be learned in an explicit (and possibly deterministic) way. For example, to learn a decision tree, the training set is used to determine what nodes are split and what condition is applied [9]. Consequently, when making bagging ensembles of decision trees (i.e., random forests [8]), one needs to build each decision tree as a function of the respective \mathbb{T}_j , and so a multiplicative β term in the asymptotics cannot be avoided. The situation is different in GP. In GP, the structure of an individual emerges as an implicit byproduct of the whole evolutionary process; Fitness evaluation, in particular, is not responsible for altering structure. We exploit this.

Recall that each \mathbb{T}_j is obtained by bootstrap of the original \mathbb{T} , thus contains only elements of \mathbb{T} . It follows that an individual's output computed over the observations of \mathbb{T}_j contains only elements that are also elements of the output computed over \mathbb{T} . So, if we compute the output over \mathbb{T} , we obtain the output elements for $\mathbb{T}_j, \forall j$. Formally, let \mathbb{S}_j be the collection of indices

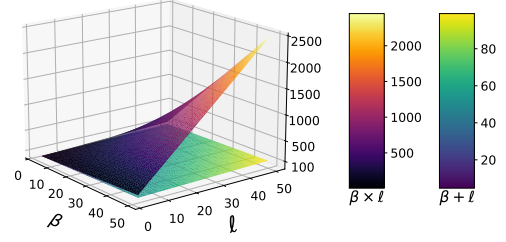


Figure 1: Scaling (vertical axis) of $\beta \times \ell$ and $\beta + \ell$.

that identifies \mathbb{T}_j , i.e., $\mathbb{S}_j = [s_1^j, \dots, s_n^j]$ s.t. $s_i^j \in \{1, \dots, n\}$ and $\{(\mathbf{x}_k, y_k)\}_{k \in \mathbb{S}_j} = \{(\mathbf{x}_k, y_k)\}_{k=s_i^j}^{s_n^j} = \mathbb{T}_j$. Then one can:

- (1) Compute *once* the output of the estimator over \mathbb{T} , i.e., $\{o_i\}_{i=1}^n$;
- (2) For $j = 1, \dots, \beta$, assemble a \mathbb{T}_j -specific output $\{o_k\}_{k \in \mathbb{S}_j}$ from $\{o_i\}_{i=1}^n$;
- (3) For $j = 1, \dots, \beta$, compute $\text{Loss}(\{y_k\}_{k \in \mathbb{S}_j}, \{o_k\}_{k \in \mathbb{S}_j})$ as j th fitness value.

Step 1 costs $O(\ell n)$, step 2 and step 3 cost $O(\beta n)$, they are executed in sequence:

$$O(\ell n) + O(\beta n) + O(\beta n) = O(n(\ell + \beta)). \quad (1)$$

This method is asymptotically faster than re-computing the output over each \mathbb{T}_j whenever $\ell + \beta < \ell \beta$ —basically in any meaningful scenario. Fig. 1 shows at a glance, for growing β and ℓ , that the additive contribution $\beta + \ell$ quickly becomes orders of magnitudes better than the multiplicative one $\beta \times \ell$. Memory-wise, all we need is to store each \mathbb{S}_j at initialization, which costs $O(\beta n)$. See Sec. 7.1 for experiments.

Note that the time cost of fitness evaluation (for the entire population) normally dominates the one of selection and the larger the number of observations in the training set n , the less the cost of selection will matter. We remark that steps 2 and 3 can be implemented in terms of $(\beta \times n)$ -dimensional matrix operations, if desired (e.g., in our python implementation, we leverage numpy [65]).

Linear scaling. We can easily include linear scaling when computing the fitness on all bootstrap samples. Linear scaling is an efficient and effective method to improve the performance of GP in regression [32, 33]. It consists of computing and applying two coefficients a, b to perform an affine transformation of the output that optimally minimizes the training (optionally, root) mean squared error as in $\text{MSE}^{a,b}(y, o) = \frac{1}{n} \sum_{i=1}^n (y_i - (a + b o_i))^2$. These coefficients are:

$$a = \bar{y} - b \bar{o}, \quad b = \frac{\sum_{i=1}^n (y_i - \bar{y})(o_i - \bar{o})}{\sum_{i=1}^n (o_i - \bar{o})^2}, \quad (2)$$

where \bar{y} (resp., \bar{o}) denote the arithmetic mean of the label (resp., output) over the training set \mathbb{T} . SoTA GP algorithms often include linear scaling (or regression in some form) [38, 39, 68, 71, 76].

We incorporate linear scaling in our approach by computing β coefficients a_j, b_j to scale each \mathbb{T}_j -specific output in a similar fashion to how step 3 of the previous section is performed. This requires to add an $O(\beta n)$ term to the left-hand side of Eq. (1), which does not change the asymptotics. Implementation can again rely on matrix operations for the sake of speed (see our code).

4 EXPERIMENTAL SETUP

We attempt to (mostly) reproduce the experimental settings used in [56], to which we compare in terms of classification. Specifically, we use $n_{\text{pop}} = 500$, the selection method described in Sec. 3.2, and variation by subtree crossover and subtree mutation with equal probability (0.5). We use the uniform random depth node selection method for variation [52] to oppose bloat. If an offspring with more than 500 nodes is generated, we discard it and clone the parent.

We use ramped half-and-half initialization with tree heights 2–6 [53]. The function set is $\{+, -, \times, \div, \sqrt{\cdot}, \log\}$, with the last three operators implementing protection by, respectively, $\widetilde{\div}(a, b) := a \times \text{sign}(b)/(|b| + \varepsilon)$, $\widetilde{\sqrt{x}} := \sqrt{|x|}$, $\widetilde{\log}(x) := \log(|x| + \varepsilon)$, with $\text{sign}(0) := 1$ and $\varepsilon = 10^{-10}$. Alongside the features, we include an ephemeral random constant terminal [53] (even though [56] chose not to) with values sampled from $\mathcal{U}(-5, 5)$, because ephemeral random constants can improve performance [53, 71] (and other algorithms we compare to use them). 2SEGP needs only one additional hyperparameter compared to classic GP, i.e., the desired ensemble size β . We set $\beta = 0.1 \times n_{\text{pop}} = 50$ as a rule of thumb. We analyze other settings of β in Sec. 7.1.

We use z-score data standardization as advised in [18]. We include linear scaling for both regression and classification tasks. In our case it is plausible to apply linear scaling in classification (prior to rounding the output to the nearest class) since the considered problems are binary (the label is 0 or 1). For completeness, we also include results without linear scaling for classification.

A run consists of 100 generations. We conduct 40 independent runs to account for the randomness of both GP and training-test splitting, for which we use a 70%–30% ratio as in [56, 76]. Statistical significance is evaluated using pairwise non-parametric Mann-Whitney U tests with p -value < 0.05 and Holm-Bonferroni correction [27, 47]. In particular, we say that an algorithm is among the best ones if no other performs significantly better.

5 COMPETING ALGORITHMS AND CONSIDERED DATASETS

Classification. For comparisons on classification problems, the first set of results we consider was provided by the authors of [56]. From [56], we report the results of the best-performing ensemble algorithm “ensemble GP with weighted voting” (eGPw); the best-performing non-ensemble algorithm “Multidimensional Multiclass GP with Multidimensional Populations” (M3GP), and classic GP (cGP). M3GP in particular is a SotA GP-based feature construction approach. In [56], M3GP is found to outperform most of the other (GP and non-GP) algorithms, including random forest.

We further include our own re-implementation of “Diverse Niching Genetic Programming” (DivNichGP), made by following [73], and that we make available at <https://github.com/marcovirgolin/DivNichGP>. For DivNichGP, we maintain equal subtree crossover and mutation probability, but also allow reproduction to take place with a 5% rate, to follow the settings of the original paper. DivNichGP internally uses tournament selection; We set this to size 8 (as for our cGP for regression, described below). For DivNichGP’s niching mechanism, we use the same distance threshold of 0 and maximal niche size of 1 as in [73]. Since DivNichGP uses a validation set to aggregate the ensemble, we build a pseudo-validation set by

taking the out-of-bag observations of the last-sampled realization of the training set. All the other settings are as in Sec. 4.

The datasets we consider for classification are the five real-world datasets used in [56] that are readily available from the UCI repository [20]. We refer to [56] for details on these datasets.

Regression. For regression, we report results from [76] (see their Table 7), i.e., median test errors of SotA GP regression algorithms. These algorithms are “Evolutionary Feature Synthesis” (EFS) [1], “Genetic Programming Toolbox for The Identification of Physical Systems” (GPTIPS) [58, 59] (and a modified version mGPTIPS that uses settings similar to those of EFS), and “Geometric Semantic Genetic Programming with Reduced Trees” (GSGP-Red) [48]. We refer to [76] for the settings and choices made for these algorithms.

We further include a home-baked version of cGP that uses tournament selection of size 8 (we also experimented with size 4 and truncation selection, but they performed worse), with all other settings being as explained before. We use again our re-implementation of DivNichGP. Next, as additional ensemble learning GP algorithm, we consider the “Spatial Structure with Bootstrapped Elitism” (SS+BE) algorithm proposed in [17], by means of results that were provided by the first author of the work. The settings for SS+BE are slightly different from those of 2SEGP in that they follow those presented in [17], as prescribed by the author: Evolution runs for 250 generations, with a population of size 196, and using a 14×14 toroidal grid distribution.

Next, we consider two recent algorithms that improve variation. The first is the GP version of the Gene-pool Optimal Mixing Evolutionary Algorithm (GP-GOMEA) [70, 71]. GP-GOMEA uses a form of crossover that preserves high-mutual information patterns. Since GP-GOMEA requires relatively large population sizes to infer meaningful patterns but converges quickly, we shift resources between population size and number of generations, i.e., we set $n_{\text{pop}} = 5000$ and use only 10 generations. Moreover, GP-GOMEA uses a fixed tree template representation: We set the template height to 7 so that up to 255 nodes can be hosted (half the maximum allowed size for the other algorithms). Second and last, we include the linear scaling-enhanced version of the semantic operator Random Desired Operator [52, 75], denoted by $\text{RDO}_{+LS}^{\text{XLS}}$ in [68]. $\text{RDO}_{+LS}^{\text{XLS}}$ uses a form of semantic-driven mutation based on the internal computations of GP subtrees and a library of pre-computed subtrees. We use the traditional “population-based” library, updated every generation and storing up to 500 subtrees, up to 12 deep.

Like for 2SEGP, linear scaling (or some similar form thereof, see [76]) is also used for the other algorithms (except for GSGP-Red for which it was not used [76]). We remark that while the generational cost of 2SEGP is only marginally larger than the one of cGP (as explained in Sec. 3), the same is often not true for the competing SotA algorithms, some of which take substantially more time to run (we refer to the respective papers for details). Hence, in many comparisons, 2SEGP can be considered to be disadvantaged.

For the sake of reproducibility we rely once more on datasets used in previous work, and this time specifically on the four real-world UCI datasets of [76].

6 BENCHMARK RESULTS

Classification. Table 1 shows the accuracy obtained by eGPw, M3GP, DivNichGP, cGP, and of course 2SEGP, the latter with and without linear scaling. At training time, M3GP is significantly best on three out of five datasets, while 2SEGP is second-best. Compared to eGPw and DivNichGP, which also evolve ensembles, 2SEGP performs better (on Heart significantly so), except for on Parks when linear scaling is disabled. This is the only dataset where we observe a substantial drop in performance when linear scaling is turned off. When testing, due to the generalization gap and the Holm-Bonferroni correction, less results are significantly different. This is evident for BCW. Compared to M3GP, 2SEGP is significantly better on Parks, but worse on Sonar. On Heart, M3GP is no longer superior, as substantial performance is lost when testing. Note also that DivNichGP, possibly because it uses a (pseudo-)validation set to choose the final ensemble, exhibits slightly (but not significantly) better generalization than 2SEGP on Heart and Iono. Overall, despite being simpler, 2SEGP fares well against DivNichGP, eGPw, and even M3GP.

Regression. Table 2 shows the results of 2SEGP, DivNichGP, SS+BE, GP-GOMEA, RDO_{+LS}^{xLS} , cGP, and the algorithms from [76] (only test is reported in their Table 7) in terms of Root Mean Squared Error (RMSE). 2SEGP always outperforms DivNichGP with the exception of training on ENH and testing on ENC. Similarly, 2SEGP outperforms SS+BE on almost all cases (not when testing on ENC). 2SEGP is also competitive with the SotA algorithms, as it is only significantly worse than GP-GOMEA and RDO_{+LS}^{xLS} on ENH when testing. On ASN, 2SEGP is not matched by any other algorithm. Interestingly, our implementation of cGP achieves rather good results on most datasets, and performs better in terms of median RMSE than some of the SotA algorithms from [76].

7 INSIGHTS

In this section, we provide insights about our proposal. We begin by assessing the role of β in terms of prediction error and time, including when the ensemble is formed by an SIEL-App. Next, we investigate our selection method by ablation. Last but not least, we peek into the effect of classic GP variation in 2SEGP. From now on, we consider the regression datasets.

7.1 On the role of the ensemble size β

We assess the performance gain (or loss) of the approach when β is increased while the population size n_{pop} is kept fixed. We include a comparison to obtaining an ensemble by running independent cGP evolutions, i.e., as in a classic SIEL-App. For 2SEGP, we scale β (approx.) exponentially, i.e., $\beta = 5, 25, 50, 100, 250, 500$. For our SIEL-App, we use $\beta = 1, 2, \dots, 10$, as running times of sequential executions quickly become prohibitive. We include cGP, DivNichGP, and SS+BE in the comparison. All settings are as before (Sec. 4).

Role of β in 2SEGP. Fig. 2 shows the distribution of test RMSE against the average time taken when using different β settings (results on the training set are not shown here but follow the same trends). For now we focus on 2SEGP (red crosses), and will consider the other algorithms later. Larger ensembles seem to perform

similarly to, or slightly better than, smaller ensembles, with diminishing returns. Statistical tests between pairwise configurations of β for 2SEGP reveal that most test RMSE distributions are not significantly different from each other (p -value ≥ 0.05 except, e.g., between $\beta = 5$ and $\beta = 500$ on CCS; and between $\beta = 5$ and $\beta = 250$ on ENH). In particular, we cannot refute the null hypothesis that larger β leads to better performance because inter-run performance variability is relatively large (this is in part due to performance loss when testing). Hence, setting β to large values such as $\beta = 1.0 \times n_{pop}$ results in a time cost increase for no marked performance gain.

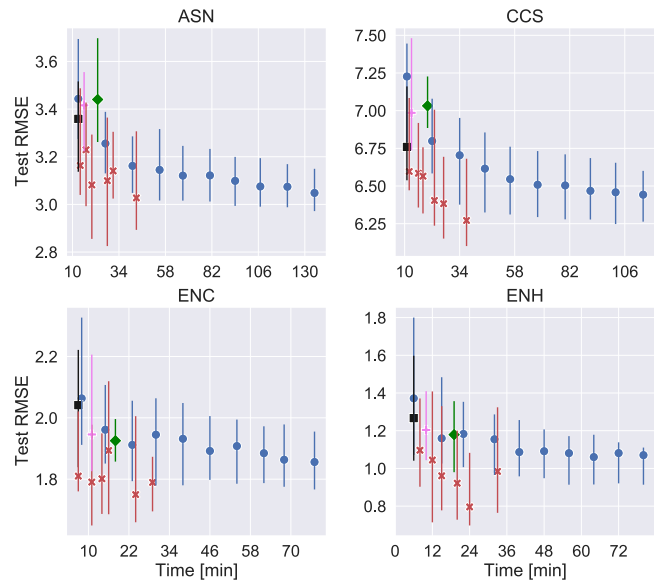


Figure 2: Distribution of test RMSE (median and interquartile range) w.r.t. average time taken by 2SEGP (in red), our SIEL-App (in blue), DivNichGP (in green), and SS+BE (in pink); or a single estimator by cGP (in black). For 2SEGP, β is scaled approximately exponentially (from left to right, β is 5, 25, 50, 100, 250, 500). For our SIEL-App, β is scaled linearly (from left to right, β is 1, 2, 3, \dots , 10).

Delving deeper, Fig. 3 shows, for different β settings in ASN runs, how many individuals are different from one another during the evolution (with $n_{pop} = 500$). This is shown for the ensemble, i.e., the collection of β individuals that are elite according to a \mathbb{T}_j -specific fitness value, and for the population. We consider *exact syntactical copies* (rather than, e.g., on semantic equivalence) to better assess the influence of selection, which copies individuals as they are. The plots on the left show that, no matter how big β is, only a very small number of distinct individuals are top-ranking across all the bootstrap samples at initialization. As the evolution proceeds, the larger β is, the more the ensemble will be redundant. The bottom-left plot shows that, no matter what β is (excl. $\beta = 5$), one-fifth of the final ensemble is made by duplicates of one type of individual.

The plots on the right show how β affects the population. We know that, in classic GP, duplicates can rapidly increase in early stages to then decrease later, when small modifications of a same root structure are generated [10, 11, 42]. This effect can be seen

Table 1: Median accuracy (higher is better) \pm interquartile range of 2SEGP, 2SEGP w/o linear scaling (w/oLS), DivNichGP, eGPw, M3GP, and cGP on the UCI datasets of [56]. Underlined results are best, i.e., not significantly worse than any other.

Algorithm	Training					Test				
	BCW	Heart	Iono	Parks	Sonar	BCW	Heart	Iono	Parks	Sonar
2SEGP (ours)	<u>0.995</u> \pm 0.005	0.944 \pm 0.022	<u>0.976</u> \pm 0.017	0.948 \pm 0.011	0.966 \pm 0.034	<u>0.965</u> \pm 0.018	<u>0.815</u> \pm 0.062	<u>0.896</u> \pm 0.047	<u>0.936</u> \pm 0.012	0.738 \pm 0.067
w/oLS (ours)	0.995 \pm 0.006	0.947 \pm 0.021	0.978 \pm 0.012	0.892 \pm 0.021	0.959 \pm 0.036	<u>0.965</u> \pm 0.013	0.809 \pm 0.049	0.896 \pm 0.047	0.885 \pm 0.031	0.754 \pm 0.067
DNGP	0.979 \pm 0.010	0.915 \pm 0.021	0.955 \pm 0.015	0.931 \pm 0.057	0.924 \pm 0.043	<u>0.959</u> \pm 0.019	<u>0.815</u> \pm 0.049	<u>0.901</u> \pm 0.026	0.917 \pm 0.055	0.730 \pm 0.063
eGPw	0.983 \pm 0.008	0.907 \pm 0.025	0.884 \pm 0.032	0.923 \pm 0.042	0.924 \pm 0.034	<u>0.956</u> \pm 0.018	<u>0.790</u> \pm 0.034	<u>0.830</u> \pm 0.057	0.822 \pm 0.064	0.762 \pm 0.060
M3GP	0.971 \pm 0.002	<u>0.970</u> \pm 0.017	0.932 \pm 0.042	<u>0.981</u> \pm 0.024	<u>1.000</u> \pm 0.012	<u>0.957</u> \pm 0.014	<u>0.778</u> \pm 0.069	<u>0.871</u> \pm 0.057	0.897 \pm 0.051	<u>0.810</u> \pm 0.071
cGP	0.964 \pm 0.016	0.825 \pm 0.033	0.773 \pm 0.060	0.842 \pm 0.077	0.769 \pm 0.055	<u>0.961</u> \pm 0.018	0.784 \pm 0.049	0.745 \pm 0.057	0.797 \pm 0.102	0.714 \pm 0.044

Table 2: Median RMSE (smaller is better) \pm interquartile range of the considered algorithms on the UCI datasets of [76]. Median results of GPTIPS, mGPTIPS, EFS, and GSGP-Red are reported from [76]. Underlined results are best, i.e., not significantly worse than any other (excl. the algs. from [76] as we only have medians). Best median-only test results are starred.

Algorithm	Training				Test			
	ASN	CCS	ENC	ENH	ASN	CCS	ENC	ENH
2SEGP (ours)	2.899 \pm 0.290	<u>5.822</u> \pm 0.353	<u>1.606</u> \pm 0.200	<u>0.886</u> \pm 0.556	<u>3.082</u> \pm 0.438	<u>6.565</u> \pm 0.439	<u>1.801</u> \pm 0.263	<u>0.961</u> \pm 0.553
DivNichGP	3.360 \pm 0.343	6.615 \pm 0.454	1.809 \pm 0.190	<u>1.079</u> \pm 0.415	3.458 \pm 0.487	7.031 \pm 0.370	1.930 \pm 0.156	1.158 \pm 0.398
SS+BE	3.271 \pm 0.316	6.517 \pm 0.412	1.882 \pm 0.363	1.190 \pm 0.291	3.416 \pm 0.333	6.986 \pm 0.744	1.946 \pm 0.380	1.204 \pm 0.366
GP-GOMEA	3.264 \pm 0.172	6.286 \pm 0.300	<u>1.589</u> \pm 0.079	<u>0.739</u> \pm 0.138	3.346 \pm 0.238	<u>6.777</u> \pm 0.313	<u>1.702</u> \pm 0.200	<u>0.804</u> \pm 0.184
RDO ^{xLS} _{+LS}	3.482 \pm 0.172	6.476 \pm 0.249	1.703 \pm 0.125	<u>0.819</u> \pm 0.186	3.579 \pm 0.245	6.800 \pm 0.423	<u>1.791</u> \pm 0.180	<u>0.881</u> \pm 0.309
cGP	3.160 \pm 0.295	6.279 \pm 0.305	1.851 \pm 0.441	1.196 \pm 0.431	3.359 \pm 0.379	<u>6.759</u> \pm 0.623	<u>2.041</u> \pm 0.383	1.267 \pm 0.556
GPTIPS	-	-	-	-	4.138	8.762	2.907	2.538
mGPTIPS	-	-	-	-	4.003	7.178	2.278	1.717
EFS	-	-	-	-	3.623	6.429*	1.640*	0.546*
GSGP-Red	-	-	-	-	12.140	8.798	3.172	2.726

for small β values. For (too) large β values, a single generation is sufficient to annihilate population diversity. This is because our selection causes top-ranking individuals across the \mathbb{T}_j s to get β copies, and at initialization only a few individuals have decent performance. Nevertheless, considering Fig. 2, this does not seem to break the algorithm in terms of test RMSE. This could be explained by the fact that larger β values also allow for larger diversity gains later on, as visible in the last generations of the top-right plot. In fact, for large β there are many \mathbb{T}_j s and thus a larger number of elites is maintained. Conversely, when β is smaller (e.g., 5 or 25), less elites are possible and population diversification caps sooner.

Since many ensemble members can be duplicates, we can prune the ensemble obtained at the end of the run. In fact, we remark that if one (takes a weighted average of the linear scaling coefficients shared by duplicate individuals and) removes duplicates, the ensemble retains the same predictive power. Pruning for $\beta = 50$ already results in considerable reductions of the ensemble size, between 34% (for ENH) and 75% (for CCS) of the original size.

Overall, these results show that: (i) Performance-wise, 2SEGP is relatively robust to the setting of β ; (ii) The ensemble may contain duplicates, but this does not represent an issue because duplicates can be trimmed off without any decrease of predictive power; and, ultimately, (iii) It is sensible to use values of β between 5% and 30% of the population size.

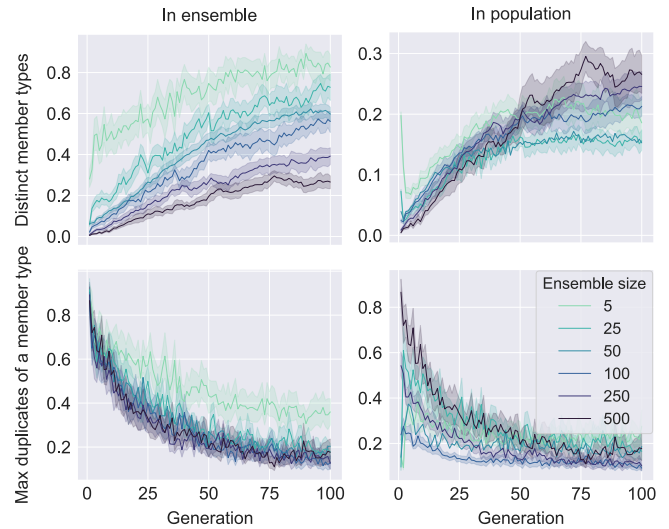


Figure 3: Mean and 95% conf. intervals of 40 runs on ASN (w. $n_{pop} = 500$) of two aspects of diversity (top and bottom) as ratios over ensemble (left) or population size (right).

Comparing with the SIEL-App. The time cost taken by our SIEL-App to form an ensemble of size β is approximately β times the time of performing a single cGP evolution, as expected (we address potential for parallelism in the last paragraph of this section). As can be seen in Fig. 2, 2SEGP can build larger ensembles in a fraction of the time taken by the SIEL-App, in line with our expectation from Eq. (1). We also report the performance of SS+BE (run on a different machine by the first author of [17]) and DivNichGP for $\beta = 50$. In brief, 2SEGP with reasonable settings (e.g., $\beta = 25, 50$) has a running time which is in the same ballpark of the times taken by SS+BE and DivNichGP, hence it is similarly efficient.

We now focus on comparing 2SEGP with the SIEL-App and start by considering the setting $\beta = 5$ for both, i.e., in each plot, the first red point and the fifth blue point, respectively. Interestingly, while 2SEGP uses only a fraction of the computational resources required to learn the ensemble compared to our SIEL-App, the ensembles obtained by the SIEL-App do not outperform the ensembles obtained by 2SEGP. The SIEL-App starts to perform slightly better than cGP already with $\beta = 2$, but at the cost of twice the running time. Within that time, 2SEGP can use 50 bootstrap samples (3rd red dot) and typically obtains better performance than any other algorithm. In general, given a same time limit, *under-performing runs of 2SEGP are often better than or similar to average-performing runs of the SIEL-App*, thanks to the former being capable of evolving larger ensembles. A downside of 2SEGP is that it obtains larger inter-run performance variance than the SIEL-App. Nevertheless, this is only natural because the latter uses a new population to evolve each ensemble member.

We remark that if the population size (which we now denote by $|\mathbb{P}|$ for readability) and/or the number of generations (G) required by our SIEL-App are reduced as to make the SIEL-App match the computational expensiveness of 2SEGP, then the SIEL-App performs poorly. This can be expected because (cfr. Sec. 3.3):

$$\begin{aligned} \text{Time cost of the SIEL-App} &\approx \text{Time cost of 2SEGP} \\ \beta G^{\text{SIEL-App}} |\mathbb{P}|^{\text{SIEL-App}} \ell n &\approx G^{\text{2SEGP}} |\mathbb{P}|^{\text{2SEGP}} n(\beta + \ell) \\ G^{\text{SIEL-App}} |\mathbb{P}|^{\text{SIEL-App}} &\approx G^{\text{2SEGP}} |\mathbb{P}|^{\text{2SEGP}} \frac{\beta + \ell}{\beta \ell}. \end{aligned} \quad (3)$$

For example, if we assume $\ell = 100$, set $\beta = 50$, $G^{\text{2SEGP}} = 100$, and $|\mathbb{P}|^{\text{2SEGP}} = 500$, then a possible setting for the SIEL-App is $|\mathbb{P}|^{\text{SIEL-App}} = 100$ and $G^{\text{SIEL-App}} = 15$ (or vice versa); If we use the same settings but reduce the ensemble size to $\beta = 5$, then for the SIEL-App we have $|\mathbb{P}|^{\text{SIEL-App}} = 105$ and $G^{\text{SIEL-App}} = 100$ (or vice versa). With the former setting, we found that the SIEL-App cannot produce competitive results. With the latter setting, the SIEL-App performed better, but still significantly worse than 2SEGP and cGP on all four regression datasets.

Finally, we must consider that, when an SIEL-App is used, each ensemble member can be evolved in parallel. If, e.g., $k\beta$ computation units are available, one can evolve a β -sized ensemble using β parallel evolutions, each one parallelized on k units. Nevertheless, with 2SEGP, resources for parallelism can be fully invested into one population, which can consequently be increased in size if desired. In other words, the results shown in this section regarding performance vs. time cost could in principle be rephrased in terms of performance vs. memory cost. We leave an analysis of how an

SIEL-App and 2SEGP compare in terms of the interplay between population size and parallel compute to future work.

7.2 Ablation of selection

We now investigate whether there is merit in partitioning the population during selection, as proposed in Sec. 3.2. If partitioning is disabled, one can no longer copy top-ranking estimators according to each \mathbb{T}_j . We consider the following alternatives: (1) Survival according to truncation (Trunc) or tournament (Tourn) selection, based on the best fitness value among any \mathbb{T}_j —We call this strategy “Push further What is Best” (PWB); (2) Like the previous point, but according to worse fitness value among any \mathbb{T}_j —We call this strategy “Push What Lacks behind” (PWL). Note that also in [74] individuals are ranked according to a PWB strategy (although the fitness values do not come from bootstrap samples).

We use the same settings of Sec. 4 (incl. $\beta = 0.1 \times n_{\text{pop}}$). Table 3 shows test RMSEs obtained using our selection method and the ablated versions. It can be noted that the ablated versions perform worse than our selection method, with a few exceptions for tournament selection with size 8 on ENC or ENH. In fact, the performance of tournament selection is the closest to the one of our selection. Using PWB or PWL leads to mixed results across the datasets, except when tournament selection with size 8 is used, where PWL is always better in terms of median results. Still, the proposed selection method leads to either equal or better performance.

Table 3: Median test RMSE \pm interquartile range of our selection method and its ablations. Tournament size is 4 or 8. Underlined results are best (not sig. worse than any other).

Selection	ASN	CCS	ENC	ENH
Ours	<u>3.082</u> ± 0.438	<u>6.565</u> ± 0.439	<u>1.801</u> ± 0.263	<u>0.961</u> ± 0.553
Trunc ^{PWB}	3.727 ± 0.292	7.347 ± 0.489	2.187 ± 0.311	1.593 ± 0.449
Trunc ^{PWL}	3.689 ± 0.310	7.373 ± 0.468	2.154 ± 0.242	1.605 ± 0.310
Tourn ₄ ^{PWB}	3.527 ± 0.372	6.996 ± 0.439	1.977 ± 0.479	1.299 ± 0.302
Tourn ₄ ^{PWL}	3.569 ± 0.517	7.025 ± 0.443	1.946 ± 0.267	1.314 ± 0.402
Tourn ₈ ^{PWB}	3.440 ± 0.485	7.042 ± 0.475	1.938 ± 0.361	<u>1.137</u> ± 0.427
Tourn ₈ ^{PWL}	3.371 ± 0.338	6.896 ± 0.541	<u>1.876</u> ± 0.189	<u>1.023</u> ± 0.370

Fig. 4 shows how the fitness values of the ensemble evolve using our selection method and the two PWB ablated versions, for one random run on ASN (we do not show the average of multiple runs as run-specific trends cancel out). It can be seen that ablated truncation performs worse than the other two, and that our selection leads to the smallest RMSEs. At the same time, our selection leads to rather uniform decrease of best-found RMSEs across the bootstrap samples. Conversely, when using Tourn₄^{PWB}, some RMSEs remain large compared to the rest, e.g., notably so for \mathbb{T}_7 , \mathbb{T}_{40} , and \mathbb{T}_{47} .

These results indicate that it is important to include partitioning as to promote uniform improvement across the bootstrap samples. Since tournament selection performs rather well, and in particular better than simple truncation selection, it would be worth studying whether our selection method can be improved by incorporating tournaments in place of truncations, or SotA selection methods such as ε -lexicase selection [37, 40].

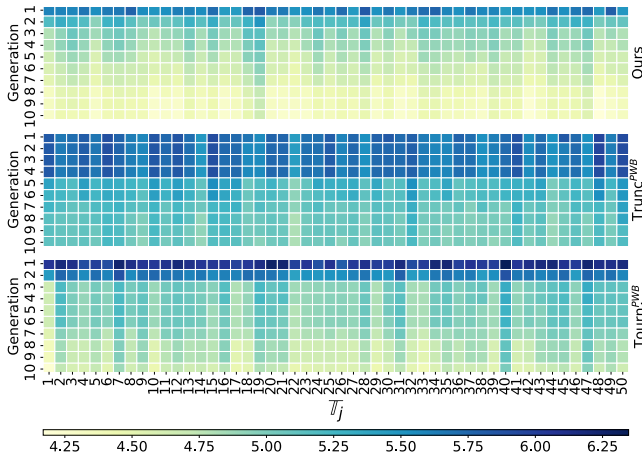


Figure 4: Training RMSEs of the best-found estimators for each T_j across 10 generations on ASN (lighter is better).

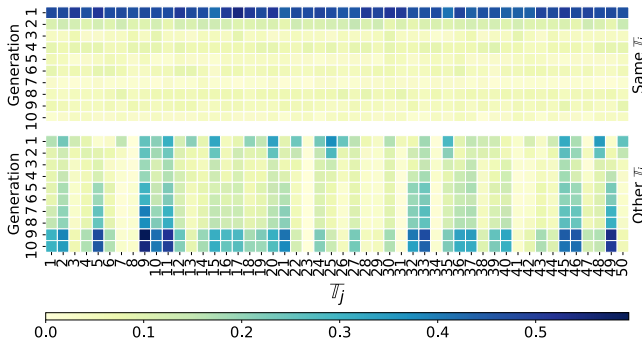


Figure 5: Frequency of producing offspring with smaller RMSE than their parents for the first 10 generations of a random run on ASN (darker is better).

7.3 Evolvability by classic variation

In our experiments, we used classic subtree crossover and subtree mutation. Our intuition was that mating between different individuals would be beneficial even if they rank better according to different bootstrap samples. To assess whether classic variation is good enough, we look at evolvability [67], here expressed as the frequency by which variation produces offspring that are fitter than their parents. We consider two aspects: (1) *Same- T_j improvement*: Frequency of producing an offspring that has a better j th fitness value than the parent; (2) *Other- T_j improvement*: Frequency of producing an offspring that has an equal or worse j th fitness value than the parent, but better k th $\neq j$ th fitness value than the parent.

Fig. 5 shows the ratios of improvement for the first 10 generations of a random run on ASN. Not only Other- T_j improvements are frequent, they can be more frequent than Same- T_j improvements (we observe the same in other runs). So, an unsuccessful variation event w.r.t. one bootstrap sample can actually be successful w.r.t. to another bootstrap sample (see, e.g., the column for T_{49}). Thus, classic variation is already able to make the population improve across different realizations of the training set. This corroborates

our proposal of leaving classic variation untouched for the sake of simplicity. Nevertheless, improvements may be possible by incorporating (orthogonal) SotA variation methods [49, 52, 71], or strategies for restricted mating and speciation [22, 46, 61].

8 CONCLUSIONS AND FUTURE WORK

We show that small changes are sufficient to make an otherwise-classic GP algorithm evolve bagging ensembles efficiently and effectively. Efficiency is a consequence of requiring only a single evolution over a single population where the nature of bootstrap sampling is exploited to perform fast fitness evaluations across all realizations of the training set. Effectiveness is perhaps somewhat surprising: The proposed algorithm can often match or even outperform state-of-the-art GP algorithms, despite being much simpler. In light of these results, we argue that GP can be considered to be naturally suited to evolve bagging ensembles, which come (almost) for free in terms of computation cost.

There are a number of avenues for future work worth exploring. Perhaps a first step could consist of studying whether it is possible to decouple selection pressure from the number of bootstrap samples. This would improve diversity preservation at the early stages of the evolution and possibly ultimately enhance ensemble quality, especially when one wishes to use a small population. Next, it will be interesting to integrate methods proposed in complex GP algorithms that are orthogonal and complementary to our approach, such as novel variation and ensemble aggregation methods. Designing “ensemble-friendly” versions of state-of-the-art selection methods (e.g., ϵ -lexicase selection [40]) could also be very beneficial, and porting knowledge from ensemble learning algorithms of different nature could lead to further improvements [54, 55]. Importantly, it would be natural to explore whether the fitness evaluation and selection changes proposed here can be applied to other types of evolutionary algorithms, e.g., to efficiently learn ensembles when optimizing the parameters or the topology of neural networks [61]. Last but not least, we remark that by learning an ensemble of many estimators, one loses an advantage of GP: The possibility to interpret the final solution [12, 26, 45, 69, 72]. Nevertheless, future work could explore integrating ensemble methods for feature importance and prediction confidence estimation [16, 35, 41], which are other relevant aspects to trust machine learning.

ACKNOWLEDGMENTS

I am deeply thankful to Nuno M. Rodrigues, João E. Batista, and Sara Silva from University of Lisbon, Lisbon, Portugal, for sharing the complete set of results of [56], and to Grant Dick from University of Otago, Dunedin, New Zealand, for providing results for SS+BE [17]. I thank Grant again, Eric Medvet from University of Trieste, Trieste, Italy, and Mattias Wahde from Chalmers University of Technology, Gothenburg, Sweden, for the insightful discussions and feedback that helped improving this paper. Part of this work was carried out on the Dutch national e-infrastructure with the support of SURF Cooperative.

REFERENCES

[1] Ignacio Arnaldo, Una-May O’Reilly, and Kalyan Veeramachaneni. 2015. Building predictive models via feature synthesis. In *Proceedings of the Genetic and*

- Evolutionary Computation Conference*. Association for Computing Machinery, 983–990.
- [2] Urvesh Bhowan, Mark Johnston, Mengjie Zhang, and Xin Yao. 2012. Evolving diverse ensembles using genetic programming for classification with unbalanced data. *IEEE Transactions on Evolutionary Computation* 17, 3 (2012), 368–386.
 - [3] Ying Bi, Bing Xue, and Mengjie Zhang. 2019. An automated ensemble learning framework using genetic programming for image classification. In *Proceedings of the Genetic and Evolutionary Computation Conference*. Association for Computing Machinery, 365–373.
 - [4] Ying Bi, Bing Xue, and Mengjie Zhang. 2020. Genetic Programming With a New Representation to Automatically Learn Features and Evolve Ensembles for Image Classification. *IEEE Transactions on Cybernetics* (2020).
 - [5] Gérard Biau. 2012. Analysis of a random forests model. *Journal of Machine Learning Research* 13, 1 (2012), 1063–1095.
 - [6] Christopher M. Bishop. 2006. *Pattern Recognition and Machine Learning*. Springer.
 - [7] Leo Breiman. 1996. Bagging predictors. *Machine Learning* 24, 2 (1996), 123–140.
 - [8] Leo Breiman. 2001. Random forests. *Machine Learning* 45, 1 (2001), 5–32.
 - [9] Leo Breiman, Jerome Friedman, Charles J. Stone, and Richard A. Olshen. 1984. *Classification and regression trees*. CRC press.
 - [10] Edmund K. Burke, Steven Gustafson, and Graham Kendall. 2004. Diversity in genetic programming: An analysis of measures and correlation with fitness. *IEEE Transactions on Evolutionary Computation* 8, 1 (2004), 47–62.
 - [11] Bogdan Burlacu, Michael Affenzeller, Michael Kommenda, Stephan Winkler, and Gabriel Kronberger. 2013. Visualization of genetic lineages and inheritance information in genetic programming. In *Proceedings of the 15th annual conference companion on Genetic and evolutionary computation*. 1351–1358.
 - [12] Alberto Cano, Amelia Zafra, and Sebastián Ventura. 2013. An interpretable classification rule mining algorithm. *Information Sciences* 240 (2013), 1–20.
 - [13] Huanhuan Chen and Xin Yao. 2010. Multiobjective neural network ensembles based on regularized negative correlation learning. *IEEE Transactions on Knowledge and Data Engineering* 22, 12 (2010), 1738–1751.
 - [14] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 785–794.
 - [15] Luzia Vidal de Souza, Aurora Pozo, Joel Mauricio Correa da Rosa, and Anselmo Chaves Neto. 2010. Applying correlation to enhance boosting technique using genetic programming as base learner. *Applied Intelligence* 33, 3 (2010), 291–301.
 - [16] Grant Dick. 2017. Sensitivity-like analysis for feature selection in genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference*. Association for Computing Machinery, 401–408.
 - [17] Grant Dick, Caitlin A. Owen, and Peter A. Whigham. 2018. Evolving bagging ensembles using a spatially-structured niching method. In *Proceedings of the Genetic and Evolutionary Computation Conference*. Association for Computing Machinery, 418–425.
 - [18] Grant Dick, Caitlin A. Owen, and Peter A. Whigham. 2020. Feature Standardisation and Coefficient Optimisation for Effective Symbolic Regression. In *Proceedings of the Genetic and Evolutionary Computation Conference*. Association for Computing Machinery, 306–314.
 - [19] Thomas G. Dietterich. 2000. Ensemble methods in machine learning. In *International Workshop on Multiple Classifier Systems*. Springer, 1–15.
 - [20] Dheeru Dua and Casey Graff. 2017. UCI Machine Learning Repository. <http://archive.ics.uci.edu/ml>
 - [21] Gianluigi Folino, Clara Pizzuti, and Giandomenico Spezzano. 2007. Mining distributed evolving data streams using fractal GP ensembles. In *European Conference on Genetic Programming*. Springer, 160–169.
 - [22] Carlos M. Fonseca and Peter J. Fleming. 1995. Multiobjective genetic algorithms made easy: selection sharing and mating restriction. In *First International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications*. IET, 45–52.
 - [23] Stanislav Fort, Huiyi Hu, and Balaji Lakshminarayanan. 2019. Deep ensembles: A loss landscape perspective. *arXiv preprint arXiv:1912.02757* (2019).
 - [24] Jerome H. Friedman. 2001. Greedy function approximation: A gradient boosting machine. *Annals of Statistics* (2001), 1189–1232.
 - [25] Jerome H. Friedman, Trevor Hastie, and Robert Tibshirani. 2000. Additive logistic regression: A statistical view of boosting. *Annals of Statistics* 28, 2 (2000), 337–407.
 - [26] Riccardo Guidotti, Anna Monreale, Salvatore Ruggieri, Franco Turini, Fosca Giannotti, and Dino Pedreschi. 2018. A survey of methods for explaining black box models. *Comput. Surveys* 51, 5 (2018), 1–42.
 - [27] Sture Holm. 1979. A simple sequentially rejective multiple test procedure. *Scandinavian Journal of Statistics* (1979), 65–70.
 - [28] Hitoshi Iba. 1999. Bagging, boosting, and bloating in genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference*. Morgan Kaufmann Publishers Inc., 1053–1060.
 - [29] Kosuke Imamura, Robert B. Heckendorn, Terence Soule, and James A. Foster. 2001. Fault-tolerant computing with N-version genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference*. Morgan Kaufmann Publishers Inc., 178–178.
 - [30] Kosuke Imamura, Robert B. Heckendorn, Terence Soule, and James A. Foster. 2002. N-version genetic programming via fault masking. In *European Conference on Genetic Programming*. Springer, 172–181.
 - [31] Ulf Johansson, Tuve Löfström, Rikard König, and Lars Niklasson. 2006. Genetically evolved trees representing ensembles. In *International Conference on Artificial Intelligence and Soft Computing*. Springer, 613–622.
 - [32] Maarten Keijzer. 2003. Improving symbolic regression with interval arithmetic and linear scaling. In *European Conference on Genetic Programming*. Springer, 70–82.
 - [33] Maarten Keijzer. 2004. Scaled symbolic regression. *Genetic Programming and Evolvable Machines* 5, 3 (2004), 259–269.
 - [34] Maarten Keijzer and Vladan Babovic. 2000. Genetic programming, ensemble methods and the bias/variance tradeoff—Introductory investigations. In *European Conference on Genetic Programming*. Springer, 76–90.
 - [35] Mark Kotanchek, Guido Smits, and Ekaterina Vladislavleva. 2008. Trustable symbolic regression models: Using ensembles, interval arithmetic and pareto fronts to develop robust and trust-aware models. In *Genetic Programming Theory and Practice V*. Springer, 201–220.
 - [36] John R. Koza. 1992. *Genetic programming: On the programming of computers by means of natural selection*. Vol. 1. MIT press.
 - [37] William La Cava, Thomas Helmuth, Lee Spector, and Jason H. Moore. 2019. A probabilistic and multi-objective analysis of lexibase selection and ϵ -lexibase selection. *Evolutionary Computation* 27, 3 (2019), 377–402.
 - [38] William La Cava and Jason H. Moore. 2020. Learning feature spaces for regression with genetic programming. In *Genetic Programming and Evolvable Machines*. Springer, 433–467.
 - [39] William La Cava, Tilak Raj Singh, James Taggart, Srinivas Suri, and Jason H. Moore. 2019. Learning concise representations for regression by evolving networks of trees. In *International Conference on Learning Representations*.
 - [40] William La Cava, Lee Spector, and Kourosh Danai. 2016. Epsilon-lexibase selection for regression. In *Proceedings of the Genetic and Evolutionary Computation Conference*. Association for Computing Machinery, 741–748.
 - [41] Balaji Lakshminarayanan, Alexander Pritzel, and Charles Blundell. 2017. Simple and scalable predictive uncertainty estimation using deep ensembles. In *Advances in Neural Information Processing Systems*. 6402–6413.
 - [42] William B. Langdon. 2017. Long-term evolution of genetic programming populations. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. 235–236.
 - [43] William B. Langdon, Steven J. Barrett, and Bernard F. Buxton. 2002. Combining decision trees and neural networks for drug discovery. In *European Conference on Genetic Programming*. Springer, 60–70.
 - [44] William B. Langdon and Bernard F. Buxton. 2001. Genetic programming for combining classifiers. In *Proceedings of the Genetic and Evolutionary Computation Conference*. Morgan Kaufmann Publishers Inc., 66–73.
 - [45] Andrew Lensen, Bing Xue, and Mengjie Zhang. 2020. Genetic Programming for Evolving a Front of Interpretable Models for Data Visualization. *IEEE Transactions on Cybernetics* (2020).
 - [46] Ngoc Hoang Luong, Han La Poutre, and Peter A. N. Bosman. 2014. Multi-objective gene-pool optimal mixing evolutionary algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference*. Association for Computing Machinery, 357–364.
 - [47] Henry B. Mann and Donald R. Whitney. 1947. On a test of whether one of two random variables is stochastically larger than the other. *Annals of Mathematical Statistics* (1947), 50–60.
 - [48] Joao Francisco B. S. Martins, Luiz Otavio V. B. Oliveira, Luis F. Miranda, Felipe Casadei, and Gisele L. Pappa. 2018. Solving the exponential growth of symbolic regression trees in geometric semantic genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference*. Association for Computing Machinery, 1151–1158.
 - [49] Alberto Moraglio, Krzysztof Krawiec, and Colin G. Johnson. 2012. Geometric semantic genetic programming. In *International Conference on Parallel Problem Solving from Nature*. Springer, 21–31.
 - [50] David Opitz and Richard Maclin. 1999. Popular ensemble methods: An empirical study. *Journal of Artificial Intelligence Research* 11 (1999), 169–198.
 - [51] Gregory Paris, Denis Robilliard, and Cyril Fonlupt. 2001. Applying boosting techniques to genetic programming. In *International Conference on Artificial Evolution (Evolution Artificielle)*. Springer, 267–278.
 - [52] Tomasz P. Pawlak, Bartosz Wieloch, and Krzysztof Krawiec. 2014. Semantic backpropagation for designing search operators in genetic programming. *IEEE Transactions on Evolutionary Computation* 19, 3 (2014), 326–340.
 - [53] Riccardo Poli, William B. Langdon, Nicholas F. McPhee, and John R. Koza. 2008. *A field guide to genetic programming*. Lulu.com.
 - [54] Philipp Probst and Anne-Laure Boulesteix. 2017. To tune or not to tune the number of trees in random forest. *Journal of Machine Learning Research* 18, 1 (2017), 6673–6690.
 - [55] Philipp Probst, Marvin N. Wright, and Anne-Laure Boulesteix. 2019. Hyperparameters and tuning strategies for random forest. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 9, 3 (2019), e1301.

- [56] Nuno M. Rodrigues, João E. Batista, and Sara Silva. 2020. Ensemble Genetic Programming. In *European Conference on Genetic Programming*. Springer, Cham, 151–166.
- [57] Stefano Ruberto, Valerio Terragni, and Jason H. Moore. 2020. SGP-DT: Semantic Genetic Programming Based on Dynamic Targets. In *European Conference on Genetic Programming (Part of EvoStar)*. Springer, 167–183.
- [58] Dominic P. Searson. 2015. GPTIPS 2: An open-source software platform for symbolic data mining. In *Handbook of Genetic Programming Applications*. Springer, 551–573.
- [59] Dominic P. Searson, David E. Leahy, and Mark J. Willis. 2010. GPTIPS: An open source genetic programming toolbox for multigene symbolic regression. In *Proceedings of the International Multiconference of Engineers and Computer Scientists*, Vol. 1. Citeseer, 77–80.
- [60] Dominik Sobania and Franz Rothlauf. 2018. CovSel: A new approach for ensemble selection applied to symbolic regression problems. In *Proceedings of the Genetic and Evolutionary Computation Conference*. Association for Computing Machinery, 529–536.
- [61] Kenneth O. Stanley and Risto Miikkulainen. 2002. Evolving neural networks through augmenting topologies. *Evolutionary Computation* 10, 2 (2002), 99–127.
- [62] Thilo Strauss, Markus Hanselmann, Andrej Junginger, and Holger Ulmer. 2017. Ensemble methods as a defense to adversarial perturbations against deep neural networks. *arXiv preprint arXiv:1709.03423* (2017).
- [63] Marco Tomassini. 2006. *Spatially structured evolutionary algorithms: Artificial evolution in space and time*. Springer.
- [64] Cao Truong Tran, Mengjie Zhang, Bing Xue, and Peter Andreae. 2018. Genetic Programming with Interval Functions and Ensemble Learning for Classification with Incomplete Data. In *Australasian Joint Conference on Artificial Intelligence*. Springer, 577–589.
- [65] Stefan Van Der Walt, S. Chris Colbert, and Gael Varoquaux. 2011. The NumPy array: A structure for efficient numerical computation. *Computing in Science & Engineering* 13, 2 (2011), 22.
- [66] Kalyan Veeramachaneni, Ignacio Arnaldo, Owen Derby, and Una-May O'Reilly. 2015. FlexGP. *Journal of Grid Computing* 13, 3 (2015), 391–407.
- [67] Sébastien Verel, Philippe Collard, and Manuel Clergue. 2003. Where are bottlenecks in NK fitness landscapes?. In *IEEE Congress on Evolutionary Computation*, Vol. 1. IEEE, 273–280.
- [68] Marco Virgolin, Tanja Alderliesten, and Peter A. N. Bosman. 2019. Linear Scaling with and within Semantic Backpropagation-Based Genetic Programming for Symbolic Regression. In *Proceedings of the Genetic and Evolutionary Computation Conference*. Association for Computing Machinery, 1084–1092.
- [69] Marco Virgolin, Tanja Alderliesten, and Peter A. N. Bosman. 2020. On explaining machine learning models by evolving crucial and compact features. *Swarm and Evolutionary Computation* 53 (2020), 100640.
- [70] 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*. Association for Computing Machinery, 1041–1048.
- [71] Marco Virgolin, Tanja Alderliesten, Cees Witteveen, and Peter A. N. Bosman. 2020. Improving Model-based Genetic Programming for Symbolic Regression of Small Expressions. *Evolutionary Computation* 0, ja (2020), 1–27. https://doi.org/10.1162/evco_a_00278 arXiv:https://doi.org/10.1162/evco_a_00278 PMID: 32574084.
- [72] Marco Virgolin, Andrea De Lorenzo, Eric Medvet, and Francesca Randone. 2020. Learning a Formula of Interpretability to Learn Interpretable Formulas. In *International Conference on Parallel Problem Solving from Nature*. Springer, 79–93.
- [73] Shaolin Wang, Yi Mei, and Mengjie Zhang. 2019. Novel ensemble genetic programming hyper-heuristics for uncertain capacitated arc routing problem. In *Proceedings of the Genetic and Evolutionary Computation Conference*. Association for Computing Machinery, 1093–1101.
- [74] Yu-Wei Wen and Chuan-Kang Ting. 2016. Learning ensemble of decision trees through multifactorial genetic programming. In *IEEE Congress on Evolutionary Computation*. IEEE, 5293–5300.
- [75] 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*. Association for Computing Machinery, 1013–1020.
- [76] Jan Žegklitz and Petr Pošík. 2020. Benchmarking state-of-the-art symbolic regression algorithms. *Genetic Programming and Evolvable Machines* (2020), 1–29.
- [77] Yifeng Zhang and Siddhartha Bhattacharyya. 2004. Genetic programming in classifying large-scale data: An ensemble method. *Information Sciences* 163, 1-3 (2004), 85–101.