Slim_gsgp: A Python Library for Non-Bloating GSGP

Liah Rosenfeld
lrosenfeld@novaims.unl.pt
NOVA Information Management School, Universidade Nova de Lisboa, Portugal

Davide Farinati
dfarinati@novaims.unl.pt
NOVA Information Management School, Universidade Nova de Lisboa, Portugal

Diogo Rasteiro
drsteiro@novaims.unl.pt
NOVA Information Management School, Universidade Nova de Lisboa, Portugal

Gloria Pietropolli
gloria.pietropolli@phd.units.it
Department of Mathematics, Informatics, and Geosciences, University of Trieste, Italy

Karina Brotto Rebuli
karina.brottorebuli@unito.it
Department of Veterinary Sciences, University of Torino, Italy

Sara Silva
sgsilva@fc.ul.pt
LASIGE, Department of Informatics, Faculty of Sciences, University of Lisbon, Portugal

Leonardo Vanneschi
lvanneschi@novaims.unl.pt
NOVA Information Management School, Universidade Nova de Lisboa, Portugal

Abstract

This paper presents `slim_gsgp`: an open-source `Python` library that provides the first ever framework for the Semantic Learning algorithm based on Inflate and deflate Mutation (SLIM-GSGP). Proposed in 2024, SLIM-GSGP is a promising non-bloating variant of Geometric Semantic Genetic Programming (GSGP). `slim_gsgp` includes all existing SLIM-GSGP variants, as well as traditional GSGP and standard Genetic Programming (GP), facilitating comparative analysis and benchmarking. Additionally, `slim_gsgp`'s parallel computation and semi-modular architecture renders it not only fast but also user-friendly and easily extensible, thereby serving as a valuable resource for researchers aiming to advance this emerging and promising area of research. The source code and documentation can be accessed at https://github.com/DALabNOVA/slim.

Keywords

SLIM-GSGP, Geometric Semantic Genetic Programming, Open Source Library, Extensibility, Python.

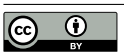
ACM Reference Format:
Liah Rosenfeld, Davide Farinati, Diogo Rasteiro, Gloria Pietropolli, Karina Brotto Rebuli, Sara Silva, and Leonardo Vanneschi. 2025. Slim_gsgp: A Python Library for Non-Bloating GSGP. In *Genetic and Evolutionary Computation Conference (GECCO '25), July 14–18, 2025, Malaga, Spain.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3712256.3726398

1 Introduction

Geometric Semantic Genetic Programming (GSGP) [1] is a variant of Genetic Programming (GP) [2] that owes its success to its ability to induce a unimodal error surface (i.e. an error surface without any local optimum) for any supervised learning problem. This grants GSGP a remarkable optimization capability, often also leading to an interesting generalization performance. Due to this feature, GSGP frequently produces highly accurate predictive models and it has been successful in many application areas in the last decade [3–8] . However, GSGP has a significant limitation: its genetic operators always create individuals that are larger than their parents. As a consequence, the final model produced by GSGP is often extremely large, making it impractical for human interpretation or even readability [1]. Although several strategies have been proposed to control individuals' growth in GSGP (e.g. 9–11), until now, none of them had been able to generate models that are sufficiently small to be readable.

The Semantic Learning algorithm based on Inflate and deflate Mutations (SLIM-GSGP) [12, 13] is a novel variant of GSGP that overcomes this limitation. It is able to generate small models, while retaining the beneficial property of GSGP of inducing a unimodal error surface. As its name suggests, SLIM-GSGP extends traditional GSGP by incorporating, alongside the traditional GSGP mutation (referred to as inflate mutation because it produces offspring larger than their parents), a novel operator called deflate mutation, that generates smaller offspring than their parents. SLIM-GSGP currently has six different variants [13], that have often demonstrated a performance comparable to (and in some cases even better than) traditional GSGP, while consistently producing models that are orders of magnitude smaller than the ones returned by traditional GSGP and compact enough for allowing human interpretation.

Despite the existence of several efficient implementations of GSGP [14–17], given the innovative nature of SLIM-GSGP, there is currently no established software framework available that integrates this method. Developing such a framework is crucial as it facilitates further development, understanding, and an in-depth exploration of the algorithm's potential. With this in mind, we propose a `Python` library for this purpose called `slim_gsgp`, using `Pytorch`'s highly-efficient data structures and a modular design.



This work is licensed under a Creative Commons Attribution 4.0 International License.
GECCO '25, July 14–18, 2025, Malaga, Spain
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1465-8/2025/07
https://doi.org/10.1145/3712256.3726398

In addition, `slim_gsgp` offers an efficient and flexible tensor-based approach, emphasizing code readability to facilitate further development. The source code of `slim_gsgp` is available at:

https://github.com/DALabNOVA/slim,

and its documentation and tutorial can be accessed at:

https://slim-library.readthedocs.io.

Over the past decade, several studies have demonstrated that mutation is a more powerful operator for GSGP than crossover. Mutation-only GSGP often yields results comparable to or better than approaches combining both mutation and crossover [18]. Furthermore, no definition for a size-aware crossover for SLIM-GSGP has been introduced as of the date of this publication. Hence, the proposed library focus solely on mutation-based GSGP. However, due to the user-friendly and modular design of the `slim_gsgp` library, we believe it is naturally well-suited for future extensions, including the integration of new crossovers, as well as other operators and additional algorithms and functionalities.

The paper is organized as follows: Section 2 provides a comprehensive overview of GSGP and the novel SLIM-GSGP, detailing their fundamental components and operators. This is followed by Section 3, which introduces the proposed `slim_gsgp` library, outlining its architecture and the computational representation of SLIM-GSGP individuals as linked lists. Additionally, this section presents a case study that exemplifies the types of analyses facilitated by `slim_gsgp`. Section 4 provides the user with essential guidelines for expanding the `slim_gsgp` library. It offers in-depth explanation of the library's implementation and modules, along with step-by-step instructions for modifying, extending, or adapting each component to suit specific research needs. Finally, Section 5 presents the main conclusions of this work, highlighting the significance of `slim_gsgp`, its potential impact on academic research, and its current available resources, including the open-source repository and documentation.

2 Background

This section provides a concise overview of GSGP and SLIM-GSGP. It begins by outlining the fundamental mutation operator of GSGP, namely Inflate Geometric Semantic Mutation (IGSM), followed by an introduction to the investigated SLIM-GSGP variants and their novel mutation operator, Deflate Geometric Semantic Mutation (DGSM). Additionally, it covers an efficient linked-list-based representation of SLIM-GSGP individuals.

2.1 Geometric Semantic Genetic Programming

In GP, the concept of *semantics* denotes the output vector produced by a program for a specific set of inputs. GSGP [1] is a variation of GP that replaces conventional syntax-oriented genetic operators with Geometric Semantic Operators (GSOs), which provide predictable effects on the semantics of individuals and introduce geometric properties in the semantic space.

The Geometric Semantic Mutation (GSM) mutates a parent function $\mathcal{T} : \mathbb{R}^n \to \mathbb{R}$ and generates an individual $\mathcal{T}_M$ such that:

$$\mathcal{T}_M = \mathcal{T} + ms \cdot (T_{R1} - T_{R2}),$$

where T_{R1} and T_{R2} are random real-valued functions with outputs in the range $[0, 1]$, and ms represents the mutation step hyperparameter. GSM operates as a ball mutation in the semantic space, generating offspring within a sphere of radius ms centered at the parent's semantics. One of GSGP's significant advantages is its ability to create a unimodal error surface, simplifying the search by eliminating local optima. For a detailed explanation of this property, the reader is referred to [19]. Nonetheless, a drawback of traditional GSM is that it invariably produces offspring larger than their parent, leading to rapid growth in individual sizes, posing challenges in terms of efficiency and interpretability. This operator is often called "Inflate" GSM (IGSM).

2.2 SLIM-GSGP

SLIM-GSGP [12, 13] extends GSGP by retaining its property of inducing a unimodal error surface while generating more compact models. It incorporates two types of mutations: the existing IGSM and a newly introduced "Deflate" GSM (DGSM) designed to yield smaller offspring.

The validity of DGSM relies on two key observations. First, from the very definition of GSM, we have:

$$\text{GSM}(T) = T + ms \cdot (T_{R1} - T_{R2}) = T - ms \cdot (T_{R2} - T_{R1})$$

Now, given that T_{R1} and T_{R2} are both random expressions drawn from the same distribution, they are interchangeable. Hence, an equivalent expression for GSM is:

$$\text{GSM}(T) = T - ms \cdot (T_{R1} - T_{R2}).$$

This reformulation is key for generating offspring that are smaller than their parents. To convince oneself of this fact, one may consider the following example. Suppose we apply GSM, say, for three consecutive times to an individual T:

$$T'_M = T + ms \cdot (T_{R1} - T_{R2}) + ms \cdot (T_{R3} - T_{R4}) + ms \cdot (T_{R5} - T_{R6}) .$$

Applying GSM again, but this time using subtraction instead of addition, results in:

$$T''_M = T + ms \cdot (T_{R1} - T_{R2}) + ms \cdot (T_{R3} - T_{R4})$$
$$+ms \cdot (T_{R5} - T_{R6}) - ms \cdot (T_{R7} - T_{R8}) .$$

T''_M is larger than T'_M, but, knowing that a common approach in the literature [20] is to reuse random trees instead of generating new ones every time they are needed, one may, for example, reuse, say, T_{R3} and T_{R4} instead of using T_{R7} and T_{R8}, obtaining:

$$T'''_M = T + ms \cdot (T_{R1} - T_{R2}) \cancel{+ms \cdot (T_{R3} - T_{R4})}$$
$$+ms \cdot (T_{R5} - T_{R6}) \cancel{-ms \cdot (T_{R3} - T_{R4})}.$$

Now, T'''_M has a smaller genotype than its parent T'_M. Thus, the functioning of the DGSM simply consists of removing a previously added term. As for IGSM, also the effect of DGSM is to perturb the semantics of the parent within the range $[-ms, ms]$, where ms is the mutation step.

Integrating both IGSM and DGSM, SLIM-GSGP maintains GSGP's distinctive properties, while being able to effectively manage the size of the individuals, offering strong potential for the generation of interpretable models. However, as the approach is less than a year old at the time we are writing this paper, it clearly

remains in its early stages, and substantial further research is still needed to fully explore its potential. The proposed library aims to foster this research by providing the community with a robust and extensible Python framework, facilitating further exploration and development.

2.2.1 SLIM-GSGP Variants. Six variants of SLIM-GSGP are implemented in `slim_gsgp`. They were proposed in [13, 21] by combining three mutation functions that return values in $[-ms, ms]$ and two different ways of perturbing an individual to obtain ball mutation on the semantic space.

The three mutation functions returning values in $[-ms, ms]$ are defined as follows.

(1) Using two random trees T_{R1} and T_{R2}, each passed through the sigmoid function, denoted by $S(\ )$, which maps outputs to $[0, 1]$:

$$2\text{SIG} = ms \cdot (S(T_{R1}) - S(T_{R2})) .$$

This corresponds to traditional GSM.

(2) Using a single random tree T_R passed through the sigmoid function S:

$$1\text{SIG} = ms \cdot (2 \cdot S(T_R) - 1) .$$

(3) Using a single random tree T_R normalized by its absolute value:

$$\text{ABS} = ms \cdot (1 - 2/(1 + |T_R|)) .$$

The two perturbation methods proposed in [13, 21] for achieving ball mutation in the semantic space, and implemented in `slim_gsgp`, are here named using the intuitive terms of "Addition" and "Multiplication" perturbations:

(1) Addition (+): it adds or subtracts a near-zero value to generate the small perturbation needed for ball mutation. IGSM adds the 2SIG, 1SIG or ABS output, while DGSM subtracts one of the previously added terms.
(2) Multiplication ($*$): it multiplies or divides by a factor close to one, to generate the small perturbation needed for ball mutation. IGSM multiplies by $1 + \phi$, where ϕ is either 2SIG, 1SIG or ABS, while DGSM divides by one of the previosuly multiplied terms.

Combining these perturbation methods with the three mutation functions yields six SLIM-GSGP variants. Each variant is denoted as [ALGORITHM][perturbation method][mutation function]. For instance, SLIM+2SIG represents SLIM-GSGP using Addition perturbation (+) and 2SIG for ball mutation within $[-ms, ms]$.

3 The `slim_gsgp` Library

`slim_gsgp` was inspired by the idea of establishing a GP-centric framework that prioritizes simplicity and extensibility. Specifically, the library stems from two main objectives. First, to create a framework dedicated exclusively to GP, including traditional GP, GSGP and SLIM-GSGP, facilitating rigorous comparisons between GP algorithms. Second, to support the research, application, and exploration of SLIM-GSGP by utilizing Python's extensive scientific ecosystem, with widely adopted libraries such as `Pytorch`, `joblib`, and `Numpy`.

3.1 Architecture of `slim_gsgp`

The framework adopted in `slim_gsgp` has modules that are common for GP, GSGP and SLIM-GSGP, corresponding to functionalities utilized by all algorithms (e.g. Data Loader, Evaluators, etc.), and modules that are exclusive for each one of the implemented GP variants. As can be seen in Figure 1, commonly utilized modules, like fitness functions, are housed in distinct, algorithm-independent modules. Conversely, features that are unique to a particular GP variant, like individual representations and crossover and mutation genetic operators, are situated within the variant's respective module. This architectural approach facilitates the framework's customization and expansion. Furthermore, to enhance computational efficiency, the `slim_gsgp` library leverages the inherent parallelism of GP [2] while maintaining the results' consistency. Specifically, it makes use of `joblib`, a Python library for parallel computing, to parallelize the generation of individuals and the fitness evaluation process. Finally, unit tests are incorporated into `slim_gsgp`, using the `Pytest` library to enhance robustness and ensuring the correct functioning of the algorithms.

3.2 Individuals in `slim_gsgp`

The implementation of individuals in `slim_gsgp` uses the classic GP tree representation [2], where internal nodes correspond to functions or operations, and leaves represent input variables or constant values. This allows GP to evolve complex solutions for symbolic regression by modifying the tree structure through genetic operators like crossover and mutation.

To ensure an effective representation, `slim_gsgp` trees are structured as nested tuples. In each tuple, the first element is a function, followed by its inputs in sequence. Functions of varying arities can be used. Crossover and mutation operators directly modify the structure of the individuals by altering the tuples and their elements. For example, the function $f(x) = x_3 \times (x_4 + (x_2 - x_1))$ will correspond to the tuple (`multiply,` x_3`, (add,` x_4`, (subtract,` x_2`,` x_1`))`).

By default, each time a crossover or mutation operator is applied in the `slim_gsgp` library, the new GP individual is evaluated from scratch. This can also be applied to GSGP individuals, however it would result in a high computational cost, as GSGP individuals tend to grow rapidly during the evolution. A solution, proposed in [15] and implemented in `slim_gsgp`, is to store the semantics of each individual and use it to compute the semantics of the offspring. Using this approach, a GSGP individual is represented as a tuple, where the first element is the genetic operator that was used to generate that individual (either Geometric Semantic Crossover (GSC) or GSM), and the subsequent elements are the operator's inputs. For example, a GSGP individual might be represented as (`geometric_crossover,` T_1`,` T_2`,` T_{R1}`)`, which signifies that the individual has been generated by applying GSC to the parents T_1 and T_2, using the random tree T_{R1}. This representation eliminates the need to store the actual structure (genotype) of the individuals, focusing instead on their semantics and making the evaluation process faster, since new individuals do not have to be evaluated from scratch.

SLIM-GSGP individuals in the `slim_gsgp` library are implemented as linked lists, as suggested in [12], where each list item

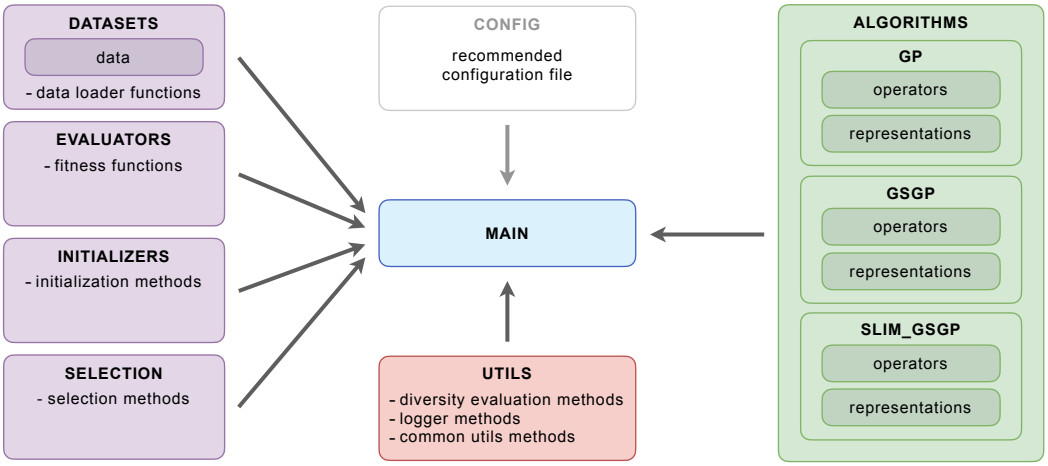


Figure 1: An overview of the `slim_gsgp` framework.

contains a subprogram. Figure 2 illustrates this representation. The

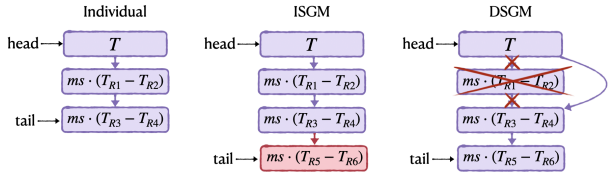


Figure 2: Example of linked list representation of a SLIM+2SIG individual, with the effects of IGSM and DGSM. IGSM appends a list item, while DGSM removes one.

first item is a nested tuple similar to a standard GP `slim_gsgp` individual. Each subsequent item of the linked list represents the new genetic material appended through IGSM, structured as a tuple `(variator, *random_trees, ms)`. DGSM modifies the individual structure by simply removing an item from the linked list. We invite the reader to consider that this operation would be much more complex if SLIM-GSGP was implemented using the traditional nested tree structures, as removing a subtree may involve modifying its child branches. A linked list of semantics is also maintained for each item of the list representing an individual. So, when IGSM or DGSM are applied, new items are either added or removed also from this list of semantic vectors. This approach speeds up the evaluation process by avoiding the need to evaluate new individuals from scratch, similarly to what happens for GSGP. For SLIM-GSGP, the semantics of the entire individual can be computed by summing or multiplying (depending on the used variant) the semantics of all items in the linked list.

Additionally, the `slim_gsgp` library includes the `reconstruct` hyperparameter for GSGP and SLIM-GSGP algorithms. This hyperparameter allows users to decide whether to save both the semantics and the structure of the individuals (in case the parameter is set to `True`) or only their semantics (parameter set to `False`). Opting to save only the semantics improves computational efficiency, making the code run faster. However, this comes with the trade-off that the user will not have the final individual genotype, which may be appropriate for interpretability, or necessary for applying the final model to unseen data. Regardless, the final individual can be obtained through rerunning the algorithm with the same seed configurations and `reconstruct` set to `True`. This will repeat the exact same evolution process, ending up with the same final individual.

Finally, the function `get_tree_representation` can be used to obtain a simplified and more readable version of a SLIM-GSGP individual, and is intended to be used at the end of the evolution, when the user wishes to extract the final model.

3.3 Case Study: Benchmarking Algorithms in `slim_gsgp`

As mentioned in Section 3, one of the primary objectives of `slim_gsgp` is to provide a robust framework that fosters research and facilitates rigorous comparisons between GP algorithms. As an example, in this Section we present a systematic comparison between the three algorithms currently available in the library, assessing their performance across a set of representative datasets. This systematic comparison was carried out using the `slim_gsgp` library, with the plots being generated from results that are easily accessible and automatically stored by the `slim_gsgp`'s logger.

Figures 3 and 4 report the best individual's performance for standard GP (referred to as STDGP in the plots), GSGP and SLIM-GSGP across various datasets, showing the evolution of the train and test RMSE as well as the size of the trees across 2000 generations, with values representing medians over 30 independent runs. A summary of the datasets utilized for this analysis is presented in Table 1. These results show that no single method is superior to all the others on all the studied problems. However, combining model's accuracy and size, we can observe some trends. First of all, the SLIM* variants generally outperform their SLIM+ counterparts, particularly

Dataset	# Features	# Samples	Reference
Concrete	8	1030	[4]
Energy	8	330	[12, 13, 22]
Istanbul	7	536	[23]
PPB	626	234	[24]
RBSP	107	372	[25]
Toxicity	626	274	[24]

Table 1: Test Problems Summary. PPB stands for Plasma Protein Binding and RBSP stands for Residential Buildings Sales Price.

in producing smaller models. Among the different SLIM* variants, SLIM*ABS and SLIM*SIG1 achieve superior balance between accuracy and model compactness. These variants consistently achieve comparable or superior accuracy to traditional approaches, while keeping model sizes that are smaller or comparable. This makes them strong candidates for further investigation, which can be effectively supported by the use of the `slim_gsgp` library.

As stated previously, one of the most important factors that distinguish SLIM-GSGP from other GSGP variants is that it permits the creation of potentially interpretable models. To corroborate this, below we report an example of an individual evolved by SLIM+SIG1 for the Istanbul test problem:

$$\text{eu} + \text{bovespa} + f\left(\frac{\text{nikkei}}{-1.0}\right) + f(\text{eu} - \text{sp}) + f(\text{bovespa} - \text{dax})$$

This model was generated after 100 generations of evolution with a population of size 100, and constrained to a maximum tree depth of 8 with both probabilities of IGSM and DGSM equal to 0.5. The meaning of the variables used by this model is documented in [23]. Function f encapsulates the SLIM+SIG1 mutation operator defined in Section 2.2.1.

4 Extensibility

In line with the primary goal of the `slim_gsgp` library (i.e., to foster research and support the exploration and development of SLIM-GSGP), this section presents the core design principles that ensure its extensibility. It also demonstrates how users can adapt and expand the library's functionality to meet a wide range of research and application needs. For more details on how to extend and contribute to `slim_gsgp`, the reader is referred to the `slim_gsgp` online documentation at https://slim-library.readthedocs.io.

Modules. The `slim_gsgp` source code is organized into six core modules: `algorithms`, `datasets`, `evaluators`, `initializers`, `selection`, and `utils`. The following subsections 4.1 to 4.6 offer developers guidance on the implementation of `slim_gsgp`, highlighting straightforward yet essential considerations to keep in mind when expanding the library.

Config files. In addition to the six core modules of `slim_gsgp`, the `config` module contains the configuration files for defining all hyperparameters used in the algorithms provided in the library. Although hyperparameters can be directly set when using the algorithms, it is highly recommended to configure them through the configuration files to ensure consistency throughout the library. Therefore, whenever a new functionality is added to any of the six core modules, the configuration files must be updated accordingly. All available hyperparameters for the algorithms are specified in the online documentation.

Unit Tests. The `slim_gsgp` repository contains a `test` directory with unit tests implemented through the `Pytest` library. Developers interested in modifying `slim_gsgp` are advised to regularly run these tests and add new ones accordingly. Of particular note are the `test_X_immutability` tests, which execute a pre-configured run of the algorithms to ensure that any changes to the general code do not modify the results. However, if the developer intend to perform this type of modifications, we advise using the `pytest.mark.skip` decorator to disable that specific test.

4.1 Adding a new algorithm

To add a new GP algorithm, for example called `new_algorithm`, to the library, it is recommended to add a directory named after the algorithm with capital letters, `NEW_ALGORITHM`, in the existing `slim_gsgp/algorithms` directory. The main `new_algorithm.py` program should be placed inside this `NEW_ALGORITHM` directory, and also named after the algorithm but using small caps. Thus, the whole path of this file would be `slim_gsgp/algorithms/NEW_ALGORITHM/new_algorithm.py`. Within `new_algorithm.py`, a class with the algorithm name should be defined, containing at least the following two methods:

- `__init__`: receives the initialization hyperparameters;
- `solve`: receives the data and solve hyperparameters. Its returned object should have an attribute or function that provides access to the trained model (for example, in GP, this is the best individual in the population at the final generation).

Additional data structures, functions and other components should be placed within the same directory, `slim_gsgp/algorithms/NEW_ALGORITHM/` in our example, with subdirectories created as needed. The existing algorithms in the library use `operators` and `representations` subdirectories to store genetic operators and algorithm-specific representation classes, respectively. It is encouraged to follow this pattern structure when applicable, though it is not mandatory. Evaluators, initializers, or selection algorithms should be added according to the instructions in next Sections. While the overall structure of the code is left to the developer's discretion, it is recommended, for compatibility with the library and efficiency, using `pytorch` library for data manipulation. Additionally, within the `slim_gsgp` directory, a `main_new_algorithm.py` should be created, containing a method named after the `new_algorithm`, accordingly. This function serves as the primary interface for users interacting with the library. It should: (i) receive all hyperparameters; (ii) validate the hyperparameters if necessary; (iii) run the initialization and solving methods; (iv) return the object that stores the resulting GP model. For consistency and ease of use, the default hyperparameters should be stored in a Python file within the `slim_gsgp/config/` directory and named accordingly to the newly created algorithm, for example `slim_gsgp/config/new_algorithm_config.py`. An optional example demonstrating how to execute the algorithm may be included in the `slim_gsgp` directory, for example named `example_new_algorithm.py`.

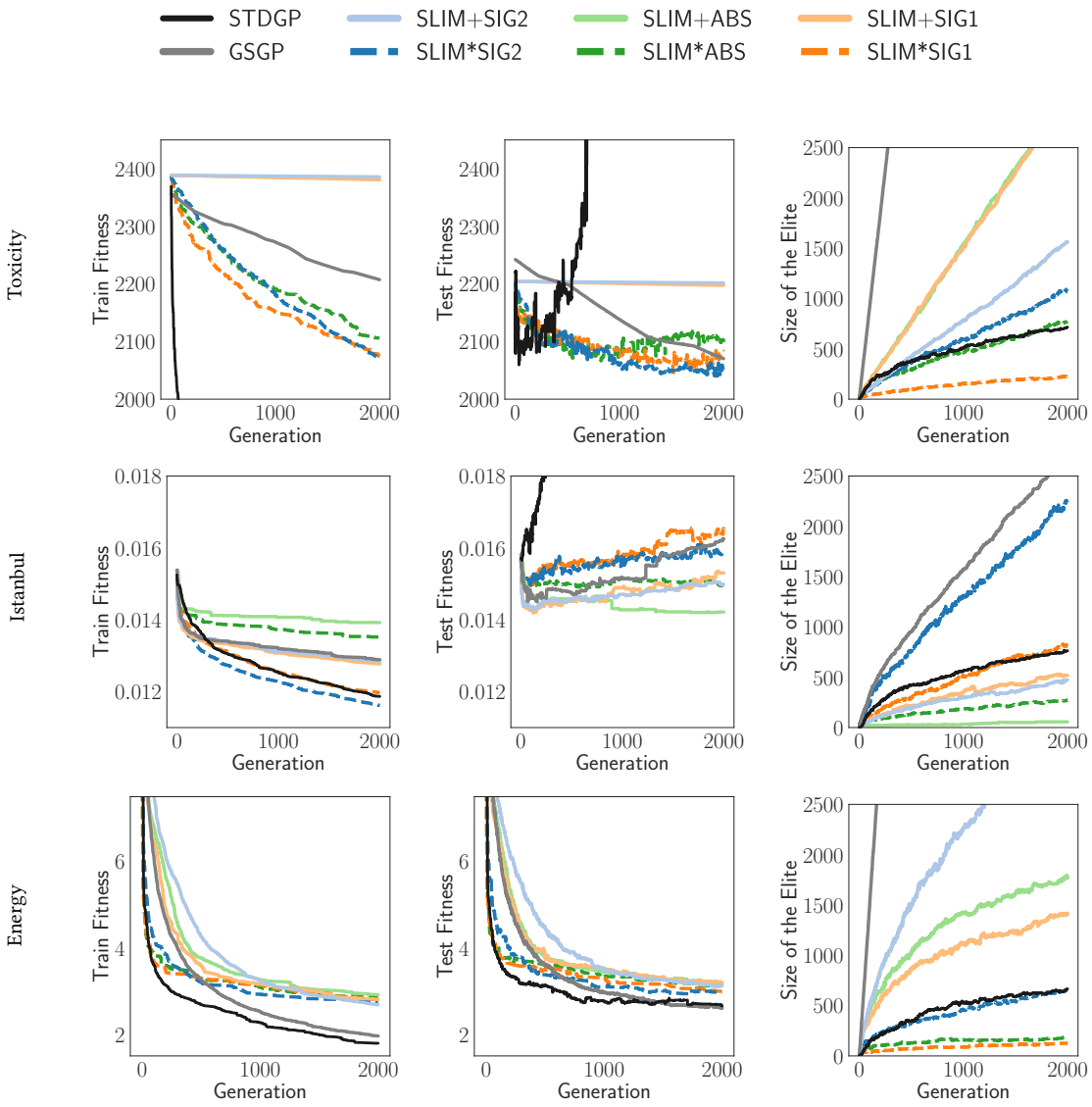


Figure 3: Evolution of the RMSE on training and test, and of the size of the elite on the Toxicity, Istanbul and Energy datasets.

4.2 Adding a new dataset

In `slim_gsgp`, datasets are `PyTorch Tensor` objects divided into features and targets. To use a custom dataset, i.e. not provided by default in the `slim_gsgp` library, developers have several options:

(1) Manual conversion: Load the data in any format to convert it into a `torch.Tensor`, splitting it into `X` (features) and `y` (target). For example, the `torch.as_tensor` method can be used for this conversion.
(2) Using `load_pandas_df` from `slim_gsgp/datasets/data/data_loader.py`. First, ensure that the dataset is loaded as a `pandas DataFrame` object, with target variable as the last column. Then, pass the `DataFrame` to `load_pandas_df` with `X_y = True`, which will handle the corresponding input features and target data.
(3) Adding a custom loader: Place the dataset in the `slim_gsgp/datasets/data` directory and implement the corresponding loading method in the `data_loader.py` file. This method should follow the format of existing loaders, including a Boolean hyperparameter `X_y` to specify whether the data should be returned as two separate tensors (`X` for features and `y` for target). Once implemented, this method can be called in the main scripts, as demonstrated in the `slim_gsgp/example_<algorithm>.py` files.

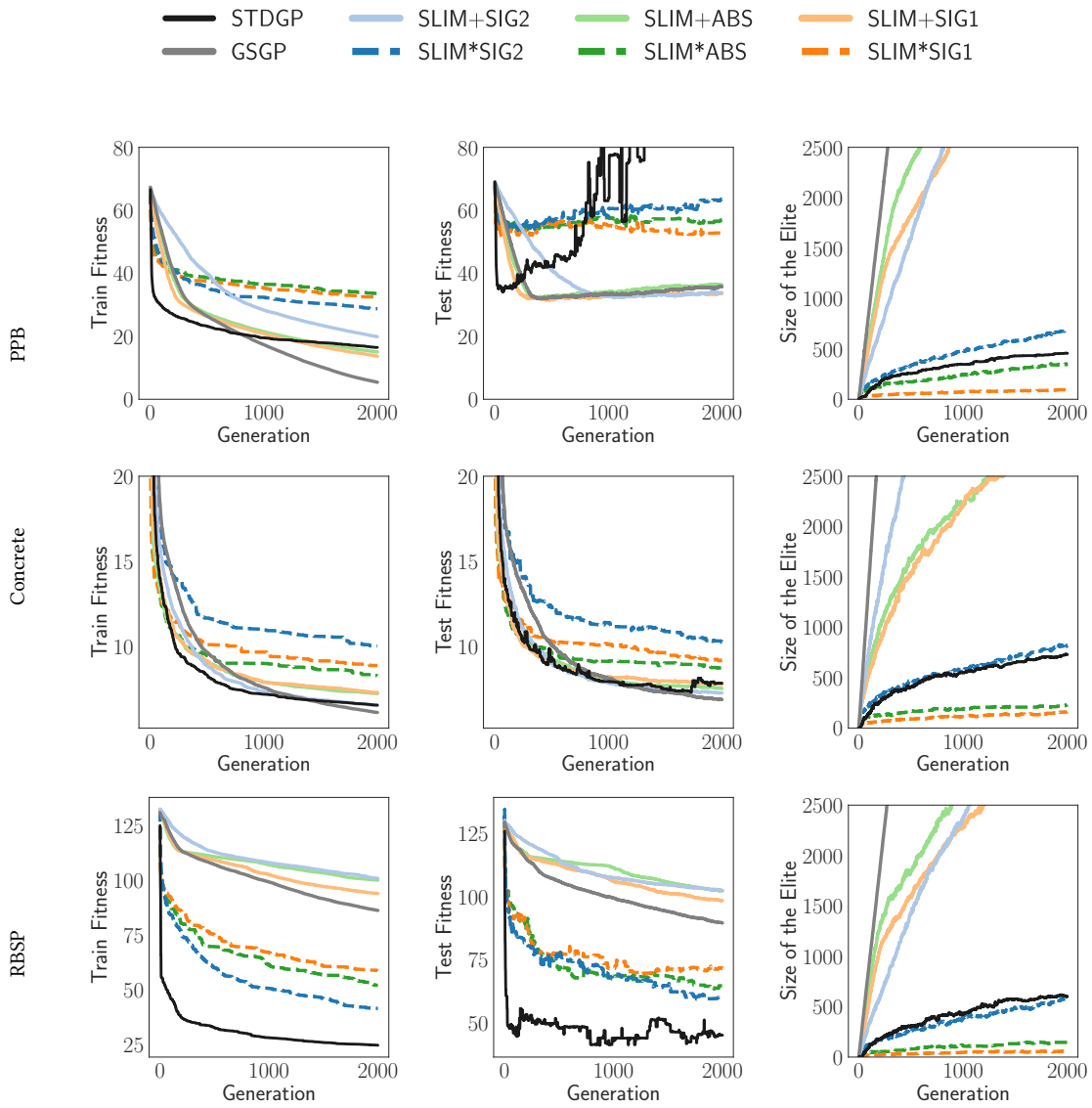


Figure 4: Evolution of the RMSE on training and test, and of the size of the elite on the PPB, Concrete and RBSP datasets.

4.3 Adding a new evaluator

Script path: `slim_gsgp/evaluators/fitness_functions`

Fitness functions must accept two one-dimensional `PyTorch Tensor` (`torch.Tensor`) objects as input: one representing the target values and the other representing the individual's predicted values. The output of the implemented evaluator should be a one-dimensional `PyTorch` Tensor containing the individual's fitness value.

4.4 Adding a new initializer

Script path: `slim_gsgp/initializers/initializers`

The functions used to initialize the population must accept at least the following six inputs: Population size, Maximum initial depth of the trees, Set of primitive functions, Set of variables, Set of constants, and Probability of selecting a constant over a variable. The output should be a Python `list` of tree-like structures represented as tuples. If additional hyperparameters are required, a nested function approach is recommended. In a nested function approach, the outer function takes the specific hyperparameters and

returns an inner function, which takes the common hyperparameters for all initializers. The sets of primitive functions, variables, and constants should be structured as `Python dictionaries`. The functions `dictionary` must use the function name as `key` and an inner `dictionary` as `value`. This inner `dictionary` must contain: (i) the "function" `key`, mappping to the function applied to the inputs; and (ii) the "arity" `key`, specifying the function arity. Variables and constants `dictionaries` must use the name of the corresponding variable or constant as its `key` and the value of the corresponding variable or constant as its `value`.

4.5 Adding a new selection algorithm

Script path: `slim_gsgp/selection/selection_algorithms`

Each selection algorithm must take as input a `Population` object and return a single individual from this `Population`. If additional hyperparameters are required, it is recommended to use a nested function, following the approach outlined for adding a new initializer. Since the majority of applications use minimizing objective functions, the default behavior implemented in the library assumes a minimization task. However, for completeness, the library also supports maximization by providing a maximization selection function and a dedicated parameter. This ensures compatibility among different scenarios, without introducing unnecessary complexity into the standard usage.

4.6 Adding a new genetic operator

When implementing a new genetic operator, it is essential to distinguish among the existing algorithms, namely GP, GSGP, SLIM-GSGP or any of the new user-defined algorithms. The newly implemented operator must be placed in the corresponding algorithm-specific directory within the `slim_gsgp/algorithms/` path, specifically under the `operators` subdirectory.

4.6.1 GP. In GP, operators take as input the *structure* (the genotype) of one or two individuals, depending on whether the operator is a mutation or a crossover. Accordingly, the genetic operators return the *structure* of the new individual(s). Noteworthy, the evaluation of the new individuals(s) is performed outside of the genetic operator function.

4.6.2 GSGP. In GSGP, operators can take as input the *individual(s) object* or the *semantics* of the individual(s), along with hyperparameters for mutation or crossover (e.g. mutation step, random tree(s), etc.). Regardless of the input type, the GSGP genetic operators must return the semantics of the offspring. Additionally, GSGP genetic operators must include the `new_data` hyperparameter, which specifies whether the semantics of the new tree needs to be recalculated from scratch on new data, or if the semantics of parent individual should be used. The latter is the standard approach during the algorithm evolution, while recompilation is typically used when generating predictions on new data. To ensure that the final prediction can be performed with the new data, the *reconstruct* option of GSGP individuals is required to be `True` during the evolutionary process.

4.6.3 SLIM-GSGP. In SLIM-GSGP, the genetic operators take as input an Individual object, which has the linked-list representation, and either add or remove an item of this list, depending on whether the operation is an inflate or deflate mutation. Both inflate and deflate operations affect the linked list of the individual's structure (`individual.collection`) and the corresponding semantics list (`individual.train_semantics`). Therefore, the mutate `Individual` object should be returned with its updated structure and semantics.

The instructions provided above for extending the library are intended as general guidelines for collaboration and development. However, depending on specific research needs, different approaches and functionalities may be required. We envision this library becoming an open and widely used resource within the research community, encouraging the continuous development and sharing of new methodologies.

5 Conclusions

We introduced `slim_gsgp`, the first Semantic Learning algorithm based on Inflate and deflate Mutations (SLIM-GSGP) modular framework built on a `Pytorch` foundation. It harnesses `Python`'s robust ecosystem to prioritize and enhance readability, accessibility and the ease of customization. This library implements all known variants of SLIM-GSGP, as well as versions of Genetic Programming (GP) and Geometric Semantic Genetic Programming (GSGP).

The first experimental results that have been obtained using the `slim_gsgp` library are encouraging: they demonstrate that SLIM-GSGP can produce compact models, while maintaining competitive or superior predictive accuracy compared to traditional GP and GSGP. This corroborates the effectiveness of the novel Deflate Geometric Semantic Mutation (DGSM), the core innovation behind SLIM-GSGP.

Designed to be intuitive, comprehensive and swift, the `slim_gsgp` library aims to support academic research and advance the understanding of SLIM-GSGP and its dynamics. The library is intended to be simple to use and effective, allowing it to be a capable tool for the whole machine learning community. To achieve this goal, we have made the source code of `slim_gsgp` available at https://github.com/DALabNOVA/slim. alongside its documentation and tutorial, which can be accessed at https://slim-library.readthedocs.io.

Acknowledgments. This work was supported by national funds through FCT (Fundação para a Ciência e a Tecnologia), under the project - UIDB/04152/2020 - Centro de Investigação em Gestão de Informação (MagIC)/NOVA IMS (DOI: 10.54499/UIDB/04152/2020) and through the LASIGE R&D Unit (UID/00408/2025).

References

[1] Alberto Moraglio, Krzysztof Krawiec, and Colin G Johnson. Geometric semantic genetic programming. In *Parallel Problem Solving from Nature-PPSN XII: 12th International Conference, Taormina, Italy, September 1-5, 2012, Proceedings, Part I 12*, pages 21–31. Springer, 2012.

[2] Riccardo Poli, William B. Langdon, and Nicholas F. McPhee. *A Field Guide to Genetic Programming*. Lulu Enterprises UK Ltd, UK, 1st edition, 2008. ISBN 978-1-4092-0073-4.

[3] Leonardo Vanneschi, Sara Silva, Mauro Castelli, and Luca Manzoni. Geometric semantic genetic programming for real life applications. In Rick Riolo, Katya

Vladislavleva, and Jason Moore, editors, *Genetic Programming Theory and Practice XI*, Genetic and Evolutionary Computation. Springer US, Computer Science Collection, 2013.

[4] Mauro Castelli, Leonardo Vanneschi, and Sara Silva. Prediction of high performance concrete strength using genetic programming with geometric semantic genetic operators. *Expert Systems with Applications*, 40(17):6856 – 6862, 2013.

[5] James McDermott, Alexandros Agapitos, Anthony Brabazon, and Michael O'Neill. Geometric semantic genetic programming for financial data. In *Applications of Evolutionary Computation: 17th European Conference, EvoApplications 2014, Granada, Spain, April 23-25, 2014, Revised Selected Papers 17*, pages 215–226. Springer, 2014.

[6] Mauro Castelli, Leonardo Trujillo, and Leonardo Vanneschi. Energy consumption forecasting using semantic-based genetic programming with local search optimizer. *Intell. Neuroscience*, 2015, jan 2015. ISSN 1687-5265. doi: 10.1155/2015/971908. URL https://doi.org/10.1155/2015/971908.

[7] Leonardo Vanneschi, Mauro Castelli, Ivo Gonçalves, Luca Manzoni, and Sara Silva. Geometric semantic genetic programming for biomedical applications: A state of the art upgrade. In *2017 IEEE Congress on Evolutionary Computation (CEC)*, pages 177–184, 2017. doi: 10.1109/CEC.2017.7969311.

[8] Francesco Marchetti, Gloria Pietropolli, Federico Julian Camerota Verdù, Mauro Castelli, and Edmondo Minisci. Automatic design of interpretable control laws through parametrized genetic programming with adjoint state method gradient evaluation. *Applied Soft Computing*, 159:111654, 2024.

[9] Joao Francisco BS Martins, Luiz Otavio VB Oliveira, Luis F Miranda, Felipe Casadei, and Gisele L Pappa. Solving the exponential growth of symbolic regression trees in geometric semantic genetic programming. In *Proceedings of the genetic and evolutionary computation conference*, pages 1151–1158, 2018.

[10] Mauro Castelli, Leonardo Vanneschi, and Aleš Popovič. Controlling individuals growth in semantic genetic programming through elitist replacement. *Computational intelligence and neuroscience*, 2016:42–42, 2016.

[11] Tomasz P Pawlak and Krzysztof Krawiec. Competent geometric semantic genetic programming for symbolic regression and boolean function synthesis. *Evolutionary computation*, 26(2):177–212, 2018.

[12] Leonardo Vanneschi. SLIM_GSGP: The non-bloating geometric semantic genetic programming. In Mario Giacobini, Bing Xue, and Luca Manzoni, editors, *Genetic Programming*, pages 125–141, Cham, 2024. Springer Nature Switzerland. ISBN 978-3-031-56957-9.

[13] Leonardo Vanneschi, Davide Farinati, Diogo Rasteiro, Liah Rosenfeld, Gloria Pietropolli, and Sara Silva. Exploring Non-bloating Geometric Semantic Genetic Programming. *Genetic Programming Theory and Practice XXI*, pages 237–258, 2 2025. doi: 10.1007/978-981-96-0077-9_12.

[14] Alberto Moraglio. An efficient implementation of gsgp using higher-order functions and memoization. In Colin Johnson, Krzysztof Krawiec, Alberto Moraglio, and Michael O'Neill, editors, *Semantic Methods in Genetic Programming*, Ljubljana, Slovenia, 13 September 2014. URL http://www.cs.put.poznan.pl/kkrawiec/smgp2014/uploads/Site/Moraglio2.pdf. Workshop at Parallel Problem Solving from Nature 2014 conference.

[15] Mauro Castelli, Sara Silva, and Leonardo Vanneschi. A c++ framework for geometric semantic genetic programming. *Genetic Programming and Evolvable Machines*, 16:73–81, 2015.

[16] Joao Francisco B. S. Martins, Luiz Otavio V. B. Oliveira, Luis F. Miranda, Felipe Casadei, and Gisele L. Pappa. Solving the exponential growth of symbolic regression trees in geometric semantic genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '18, page 1151â€"1158, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356183. doi: 10.1145/3205455.3205593. URL https://doi.org/10.1145/3205455.3205593.

[17] Mauro Castelli and Luca Manzoni. Gsgp-c++ 2.0: A geometric semantic genetic programming framework. *SoftwareX*, 10:100313, 2019. ISSN 2352-7110. doi: https://doi.org/10.1016/j.softx.2019.100313. URL https://www.sciencedirect.com/science/article/pii/S2352711019301736.

[18] Mauro Castelli, Luca Manzoni, Ivo Gonçalves, Leonardo Vanneschi, Leonardo Trujillo, and Sara Silva. An analysis of geometric semantic crossover: A computational geometry approach. In *IJCCI (ECTA)*, pages 201–208, 2016.

[19] Leonardo Vanneschi and Sara Silva. *Lectures on Intelligent Systems*. Springer, 2023.

[20] Leonardo Vanneschi, Mauro Castelli, Luca Manzoni, and Sara Silva. A new implementation of geometric semantic gp and its application to problems in pharmacokinetics. In *Genetic Programming: 16th European Conference, EuroGP 2013, Vienna, Austria, April 3-5, 2013. Proceedings 16*, pages 205–216. Springer, 2013.

[21] Illya Bakurov, José Manuel Muñoz Contreras, Mauro Castelli, Nuno Rodrigues, Sara Silva, Leonardo Trujillo, and Leonardo Vanneschi. Geometric semantic genetic programming with normalized and standardized random programs. *Genetic Programming and Evolvable Machines*, 25(1), feb 2024. ISSN 1389-2576. doi: 10.1007/s10710-024-09479-1. URL https://doi.org/10.1007/s10710-024-09479-1.

[22] Athanasios Tsanas and Angeliki Xifara. Accurate quantitative estimation of energy performance of residential buildings using statistical machine learning tools. *Energy and Buildings*, 49:560–567, 2012. ISSN 0378-7788. doi: https://doi.org/10.1016/j.enbuild.2012.03.003. URL https://www.sciencedirect.com/science/article/pii/S037877881200151X.

[23] Oguz Akbilgic, Hamparsum Bozdogan, and M. Erdal Balaban. A novel hybrid rbf neural networks model as a forecaster. *Statistics and Computing*, 24(3):365–375, May 2014. ISSN 1573-1375. doi: 10.1007/s11222-013-9375-7. URL https://doi.org/10.1007/s11222-013-9375-7.

[24] Francesco Archetti, Stefano Lanzeni, Enza Messina, and Leonardo Vanneschi. Genetic programming for computational pharmacokinetics in drug discovery and development. *Genetic Programming and Evolvable Machines*, 8(4):413–432, 2007. ISSN 1573-7632.

[25] Mohammad Rafiei. Residential Building Data Set. UCI Machine Learning Repository, 2018. DOI: https://doi.org/10.24432/C5S896.