Bingo: A Customizable Framework for Symbolic Regression with Genetic Programming

David L. Randall
u1164746@utah.edu
University of Utah
Salt Lake City, UT, USA

Tyler S. Townsend*
townsend.ts@outlook.com
Microsoft
Redmond, WA, USA

Jacob D. Hochhalter
jacob.hochhalter@utah.edu
University of Utah
Salt Lake City, UT, USA

Geoffrey F. Bomarito
geoffrey.f.bomarito@nasa.gov
NASA Langley Research Center
Hampton, VA, USA

ABSTRACT

In this paper, we introduce Bingo, a flexible and customizable yet performant Python framework for symbolic regression with genetic programming. Bingo maintains a modular code structure for simple abstraction and easily swappable components. Fitness functions, selection methods, and constant optimization methods allow for easy problem-specific customization. Bingo also maintains several features for increased efficiency such as parallelism, equation simplification, and a C++ backend. We compare Bingo's performance to other genetic programming for symbolic regression (GPSR) methods to show that it is both competitive and flexible.

CCS CONCEPTS

• **Computing methodologies** → **Genetic programming**; • **Software and its engineering** → *Abstraction, modeling and modularity*.

KEYWORDS

genetic programming, symbolic regression, genetic programming for symbolic regression

ACM Reference Format:
David L. Randall, Tyler S. Townsend, Jacob D. Hochhalter, and Geoffrey F. Bomarito. 2022. Bingo: A Customizable Framework for Symbolic Regression with Genetic Programming. In *Proceedings of The Genetic and Evolutionary Computation Conference 2022 (GECCO '22 Comanion).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3520304.3534031

1 INTRODUCTION

Genetic programming is the most common and performant approach to symbolic regression [16]. Genetic-programming-based symbolic regression (GPSR) involves generating a population of equations, creating offspring from that population, evaluating the fitness of those equations, and selecting equations to continue onto the next generation [15]. There is a wide array of methods for each step of GPSR. For generating the initial population, equations can be randomly generated, there can be a filtering process to find well-suited candidate equations, or equations can even be manually created or selected. For creating offspring, crossover or mutation may be performed, each of which have many variations. For evaluating fitness, there are several common measures of fitness and types of regression that can be performed: implicit regression [2, 10, 24, 25], physics-informed regression [13], or regression under uncertainty [1]. For selection, tournaments, elitism, niching methods [18], and pareto selection [27] are among the many possibilities.

Each of these methods has pros and cons and choosing the best method is problem dependent. While the simplicity of applying a single set of methods, regardless of context, is appealing; it is likely suboptimal to do so. We propose a GPSR framework, Bingo, which aims to solve this problem by allowing users to choose popular established methods for components of GPSR and easily define their own, if desired.

In the following sections, we will describe how Bingo approaches GPSR, give an overview of how it is structured, describe how it can be customized, and compare how it performs to other GPSR methods. While Bingo is compared to other GPSR methods, its performance is not set in stone. Bingo's main strength is its design that enables users to integrate their own code through a set of provided interfaces (Bingo is open-source and available at https://github.com/nasa/bingo). Then they may leverage any combination of Bingo-supported methods as they please. For instance, using GPSR to find an equation to a physics problem could benefit from model evaluation based on physical properties (*e.g.*, in Section 3.2) or features from other established methods (*e.g.*, Operon [3], FEAT [17], *etc.*) can be implemented for better performance on general regression problems.

2 STRUCTURE

Bingo is built upon a modular code structure with abstractions to allow for swappable components. The core components of Bingo's structure are shown in Figure 1, each of which will be detailed in the following subsections. Many of these components have multiple implementations for different problems, as shown in Figure 2. Furthermore, users can extend any of these components to create

*Views and opinions are my own and do not represent Microsoft.

GECCO '22 Comanion, July 9–13, 2022, Boston, USA

ACM ISBN 978-1-4503-9268-6/22/07…$15.00
https://doi.org/10.1145/3520304.3534031

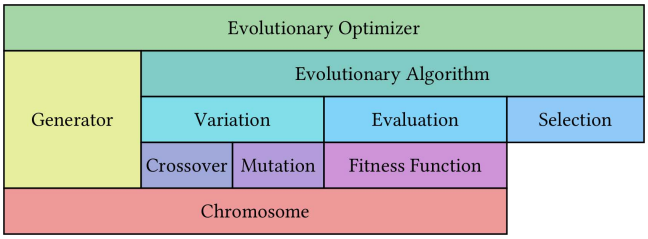

Figure 1: Classes involved in Bingo's evolutionary process.

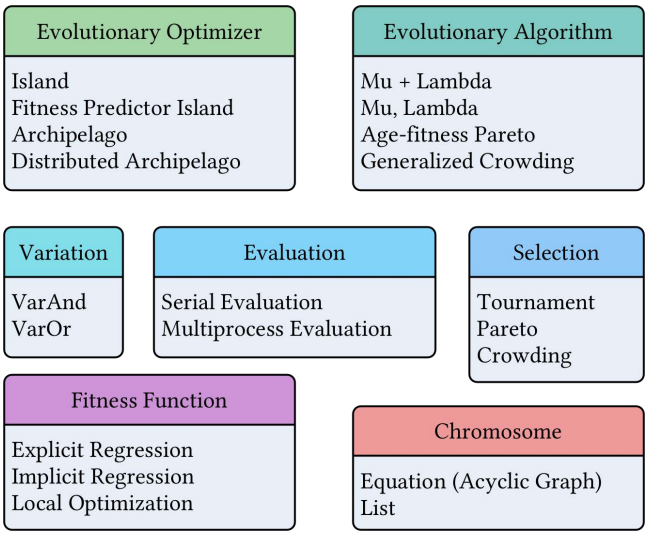

Figure 2: Implementations of Bingo's GPSR classes.

their own implementations that suit their needs. A code example of a sample Bingo configuration is shown in Appendix A. Lines of this example will be used to illustrate how to setup components described in the following sections.

2.1 Equations

Genetic algorithms and genetic programming involve the evolution of a population of individuals. In GPSR, the individuals of the population are equations [15]. In Bingo, equations are represented as instances of the chromosome object. These chromosome instances are evaluated for fitness, evolved, and selected for continuation in subsequent generations.

2.1.1 AGraph. Owing to the computational benefits over tree-encodings [23], Bingo represents equations as acyclic directed graphs (AGraphs). One such AGraph is shown in Figure 3, representing the equation $C_0X_0 + X_0 + X_1$.

Bingo's AGraphs are represented using an array of commands. As shown in Table 1, each row (command) in a command array has one node with two parameters. The first command in the command array (at $i = 0$) is a terminal load constant command. A terminal command loads a piece of data determined by its first parameter. For example, the command at $i = 0$ loads constant 0 (more on constants in Section 2.1.2). The commands at $i = 1$ and $i = 2$ are similar

Table 1: Example command array for $C_0X_0 + X_0 + X_1$

i	Node	Param. 1	Param. 2	Resulting Expression
0	constant	0	0	C_0
1	variable	0	0	X_0
2	variable	1	1	X_1
3	$*$	0	1	C_0X_0
4	$+$	1	2	$X_0 + X_1$
5	sin	2	2	$\sin(X_1)$
6	$+$	3	4	$C_0X_0 + X_0 + X_1$

except they load in variables 0 and 1, respectively. The remaining commands are operators, which perform operations on their parameters. For example, the command at $i = 3$ performs multiplication on the result of the commands at $i = 0$ and $i = 1$, resulting in C_0X_0. The commands at $i = 4$ and $i = 6$ do similar operations but with addition instead. Also note how the command at $i = 5$ is an operator that only uses one parameter, resulting in $\sin(X_1)$; operators can use one or two parameters. The last command in the array links everything together to form the equation $C_0X_0 + X_0 + X_1$ as shown in Figure 3.

Also note that not every command in the command array contributes to the final equation (*e.g.*, the command at $i = 5$). These commands are similar to vestigial structures in that they might not have a function in the current individual but can still have an impact in the evolutionary process. For example, consider the classic eye-color Punnett square, where two parents have Bb chromosomes resulting in brown eyes, with seemingly useless blue eye alleles (b in Bb). The potential offspring of these parents can have chromosomes resulting in blue eyes (bb).

2.1.2 Constants. Each equation in Bingo has the potential to have associated constants. The constants can be fixed values or placeholders for real numbers that are fitted later, for example, using continuous local optimization (CLO). CLO of constants has been shown to improve GPSR performance [7, 9, 14]. Bingo has a continuous local optimization class that can perform CLO on equations to determine optimal constant values.

2.1.3 Complexity. Complexity in Bingo is defined as the number of **utilized** nodes in the AGraph of an equation. For example, the complexity of the equation shown in Figure 3 is 6, even though a potential command array for the equation (*e.g.*, Table 1) could have more commands.

2.2 Evolutionary Optimizer

The evolutionary optimizer class, see Figure 1, handles core components of GPSR including generating the initial population and managing evolutionary steps. This high-level class allows for flexibility in coordination and execution of evolution. The main difference between an evolutionary optimizer and an evolutionary algorithm (described later) is that an evolutionary optimizer creates an initial population using the generator class. Instances of the generator class define characteristics such as the maximum number of commands equations can have, what operators are possible, *etc.* An example of an AGraph generator setup is shown on lines 23 to 29

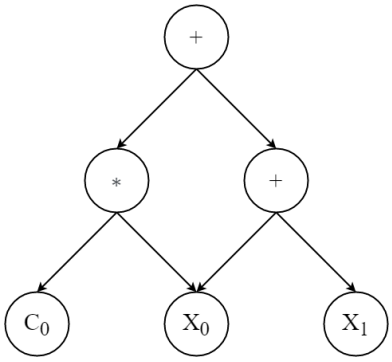

Figure 3: Example of an AGraph representing the equation $C_0X_0 + X_0 + X_1$.

of the code example (see Appendix A). Users can use the default generator as shown on line 29, or create their own to do more complex population seeding (*e.g.*, filtering, manual selection, *etc.*).

The standard GPSR evolutionary optimizer (evolution of a single population) in Bingo is called an island. An island coordinates an evolutionary algorithm and generator to create a customizable GPSR workflow (as shown on line 44 of the code example). For example, the fitness predictor island is a configuration of an island that implements coevolution of fitness predictors as described in [26]. Evolutionary optimizers can also be extended to create non-traditional GPSR workflows. For example, another evolutionary optimizer implemented in Bingo is an archipelago, which contains multiple islands and performs periodic migration between random islands [8, 19]. Users may further extend the evolutionary optimizer class to support their platform needs for execution. Bingo's parallel archipelago class demonstrates this and will be discussed later in Section 2.3.2.

2.3 Evolutionary Algorithm

While Bingo's evolutionary optimizer manages the high-level GPSR process, it has a subclass to handle the evolutionary steps of GPSR: an evolutionary algorithm. In general, an evolutionary algorithm is given a population and returns the next generation of that population. The evolutionary algorithms implemented in Bingo have three phases: variation, evaluation, and selection. Custom evolutionary algorithms may also be implemented, including those which have more/different evolutionary phases. As seen in Figure 2, there are several evolutionary algorithms implemented in Bingo for established GPSR methods including those described in [18, 27]. The setup of an evolutionary algorithm using the method described in [27] is shown on line 41 of the code example.

2.3.1 Variation. Bingo creates offspring via its variation class. The variation class has two implementations: `VarAnd` and `VarOr`. `VarAnd` creates offspring from a population by performing crossover then mutation on individuals given crossover and mutation probabilities. `VarOr` creates offspring by either performing crossover or mutation on an individual. Replication occurs when neither crossover nor mutation takes place.

Crossover of AGraphs is implemented as a single-point crossover of command arrays. Mutation of AGraphs is a random selection of either single-point command mutation, single-point operator mutation, single-point parameter mutation, prune mutation, or branch mutation. An example of setting up crossover and mutation is shown on lines 32 and 33 of the code example.

2.3.2 Evaluation. Bingo evaluates equations' fitnesses using fitness functions. Fitness functions use an equation's output on a dataset to either return a vector of fitnesses per data entry or a single fitness for the entire dataset. More specifically, fitness evaluation of a model includes a set of training data (X_i, y_i) where $i \in 1, 2, \ldots, n$ and X is an m-dimensional vector-valued input, $X_i = x_i^{(0)}, x_i^{(1)}, \ldots, x_i^{(m)}$. A model, $f : \mathbb{R}^m \to \mathbb{R}$, is then sought for these training data. For each model proposed by GPSR, $\tilde{f}$, a defined fitness function is sought to be **minimized** (Bingo treats fitness analogous to error, where the lower the error, the better). For the common case of explicit symbolic regression, a fitness would be defined as a vector:

$$F_i = \tilde{f}(X_i) - y_i \tag{1}$$

or homogenized, for example, as mean-squared error (MSE):

$$F = \frac{1}{n}\sum_{i=1}^{n}(F_i)^2. \tag{2}$$

Bingo implements several forms of fitness functions for common cases such as the vector-based function class which takes training data and a defined error metric (*e.g.*, MSE) as input and performs the requested homogenization. An example of this is shown on line 36 of the code example. Bingo also provides two main classes (derived from the vector-based function class) for explicit regression and implicit regression. The implicit regression fitness function is used for evaluating implicit equations; since the output of implicit equations are constant across a dataset, explicit regression cannot be used to judge them. Therefore, Bingo compares the partial derivatives of an equation to those of the ideal equation to get its fitness [2].

2.3.3 Selection. In GPSR, population sizes are generally fixed among generations. Therefore, individuals have to be selected from the current population and its offspring to be used for the next population. To do this, Bingo implements a selection class. Since the choice of selection method can be crucial depending on the problem [4, 12], Bingo allows its users to choose between established selection methods such as deterministic crowding [18], age-fitness [27], tournament, or create their own.

3 CUSTOMIZABILITY

Bingo utilizes its modular code structure to allow users to customize the GPSR process. This allows for turn-key access to established algorithms such as age-fitness pareto selection [27] or the creation of user-defined methods as needed. For instance, Bingo enables its users to define how the GPSR process is configured by creating custom evolutionary optimizer and evolutionary algorithm classes. This creates a flexible workflow for experimenting with high-level algorithm design decoupled from low-level intricacies.

Since it is likely the most common customization, the remainder of this section is devoted to illustrating how Bingo's fitness function class can be easily customized to suit users' needs. Bingo's fitness

function class is an abstract base class that requires only the `__call__(self, individual)` method to be implemented. This method returns the fitness of a given individual. Note that the use of `__call__` means that lambda functions can be used as drop-in replacements for fitness functions in simple cases.

3.1 Continuous Local Optimization

As mentioned in Section 2.1.2, Bingo implements CLO methods to fit constants in equations. Bingo's CLO class is implemented as a wrapper around a fitness function (either vector-valued or not) that performs optimization of constants when needed. The `__call__` method of the CLO class is illustrated here:

```
    def __call__(self, individual):
        if individual.needs_local_optimization():
            self._optimize_params(individual)
        return self._evaluate_fitness(individual)
```

The `self._evaluate_fitness(individual)` method is a wrapper around a fitness function and the `self._optimize_params(individual)` is a wrapper around a SciPy [29] optimization of constants with the fitness function as the objective. See line 37 of the code example for an example setup of the CLO object.

Other CLO methods can also be implemented in a similar fashion. For example, a custom CLO method is used in [1] to quantify uncertainty in numerical constants using Bayesian methods.

3.2 Physics-Informed Fitness

The vector-based function class can also be readily derived by the user to implement custom fitness functions. For example, consider the case of physics-informed machine learning [13] where the fitness of an individual is dependent upon its derivatives. For this, a user only needs to implement a custom class, derived from the vector-based function class, which implements a `evaluate_fitness_vector` method that takes an individual as input and returns the evaluated fitness vector. Bingo provides the `evaluate_equation_with_x_gradient_at` method for evaluation of an individual and its gradient at x. This can be used for evaluation of first derivatives: for higher-order derivatives the PyTorch [20] module is currently recommended.

As an example, consider the case of modeling the velocity of an object falling through a viscous fluid. From Newton's 2^{nd} law, the governing differential equation for acceleration, $\frac{dv}{dt}$, can be derived as $\frac{dv}{dt} = g - \frac{cv}{m}$ where v is velocity, t is time, g is acceleration due to gravity, c is a drag coefficient, and m is the mass of the object. A physics-informed fitness function could then be implemented as the residual of this equation:

```
def evaluate_fitness_vector(self, individual):
    v, dvdt = \
        individual.
     evaluate_equation_with_x_gradient_at(
        self._training_data.x)
    residual = self.g - self.c * v / self.m - dvdt
    return residual
```

This could further be augmented to simultaneously minimize the residual and match existing training data by appending elements to the returned array which represent distances from measured data to the returned array (e.g., `v - self._training_data.y`).

4 EFFICIENCY

Multiple options for increased efficiency and high-performance computing are supported within Bingo. The most prominent of these options include parallelism, coevolution of fitness predictors, equation simplification, and a C++ backend.

Bingo includes two forms of parallelism based on the desired deployment environment: distributed memory or shared memory. Parallel (distributed) evolution of islands in an archipelago is implemented using MPI4Py [6]. Parallel (shared) evaluation of individuals in an island is implemented using Python's multiprocessing tool.

Coevolution of fitness predictors has been shown to increase the efficiency and generalization of GPSR. This is achieved by using fitness predictors rather than fitness functions to evaluate equations, resulting in faster evaluations. More details regarding this method can be found in Schmidt and Lipson's paper [26].

This method is implemented in Bingo as a fitness predictor island, which is a simple, drop-in replacement for Bingo's standard evolutionary island.

Evaluation latency is tied to the complexity of equations. To reduce latency, Bingo implements algebraic simplification of equations [5] to reduce the complexity of equations during evolution. For example, $X_0+X_0+X_0+X_0+X_0$ would be simplified to $5X_0$. Bingo simplifies an equation to a simpler form and stores it alongside its original form. Then, when the equation needs to be evaluated, the simplified form is used, resulting in a faster evaluation than with the original form. However, during evolutionary processes like crossover and mutation, original forms are used to avoid removing any properties that were originally evolved.

As datasets become larger, the impact of latency for evaluating equations becomes significant. For this reason, several of the components of Bingo have been implemented in C++. This primarily serves as a backend to the Python implementations to leverage the speed of C++ while the user focuses on the high level logic of Python. The C++ backend is integrated into Bingo's Python frontend using PyBind [11]. This allows for the use of C++ objects in Python without causing significant overhead. Furthermore, once the backend is built on a user's machine, it is automatically used and can be easily disabled using flags.

5 PERFORMANCE ANALYSIS

To demonstrate how Bingo performs compared to other symbolic regression (SR) methods, its effectiveness was tested using La Cava's SRBench test suite [16]. Methods tested in this test suite are required to expose a scikit-learn [21] interface for straightforward training and prediction. Therefore, users can use Bingo's integrated scikit-learn interface for a plug-and-play experience, if desired. The test suite is separated into two portions: black-box problems and ground-truth problems. In the black-box problems, methods are judged based on the R^2 values of their predictions. In the ground-truth problems, methods are judged based on their recovery rate of ground-truth equations from data produced by those equations. Data in both types of problems were split into training and testing datasets using a 75%/25% split.

Table 2: Bingo Hyperparameter Sets

Set Number	Population Size	Stack Size
1	100	24
2	100	64
3	500	24
4	500	48
5	2500	16
6	2500	32

5.1 Black-Box Problems

For the black-box problems, each method was allowed to define a collection of hyperparameters. SR methods like Bingo were allowed to define up to six hyperparameter sets. The optimal hyperparameters per dataset were automatically chosen via five-fold halving cross-validation. Bingo was configured for these problems using the operator set {+, -, *, /, sin, cos, exp, log} with a fitness predictor island [26] using age fitness pareto selection [27]. Furthermore, Bingo used population size and stack size as the varying hyperparameters in its six hyperparameter sets as detailed in Table 2. Methods were limited to 500,000 evaluations on each dataset per hyperparameter cross-validation session or 48 hours of total training time per dataset, whichever came first.

Since GPSR methods evolve a population of equations, selecting which equation was used for testing on each dataset was an important consideration. Bingo selected the most fit individual across all generations to be used for testing.

5.1.1 Results. Figure 4 shows the tested machine learning (ML) and SR methods sorted in descending order by each method's median R^2 value on the test datasets. Methods with asterisks in front of their names are SR methods whereas the rest are other ML methods. As shown on the left plot of Figure 4, Bingo is in the top half of the methods tested based on median R^2 value. While Bingo is not the top performer of these methods based on its median R^2 value, as shown in the middle plot of Figure 4, Bingo produces very simple models (small model size) compared to the other methods tested[1]. However, the right plot of Figure 4 shows that Bingo is one of the slowest methods in training time. Bingo's slow training time is not a completely accurate depiction of its performance due to the fact that different hardware was used for Bingo's benchmark analysis. The Bingo analysis was performed locally on an Intel Xeon Gold 6230 CPU @ 2.10GHz with 4GB of memory allocated per run. Additionally, the Bingo training was performed using its pure Python implementation; performance would improve using the C++ backend

Figure 5 shows a comparison between the median model size rank and median R^2 **rank** of each method per dataset (not necessarily the same as the ranking based on median R^2 **value** as shown in Figure 4). This figure highlights how Bingo has a low median model size rank and low median R^2 rank, showing that it produces simple, yet accurate models. Other methods that perform similarly to Bingo according to these metrics are shown in the same color: Operon [3], GP-GOMEA [28], and DSR [22].

[1]For the symbolic regression methods, model size was measured as the number of nodes in an equation's graph representation (*i.e.* how Bingo defines complexity). For the other machine learning methods, other comparable metrics were used (*e.g.*, sum of nodes per tree in the RandomForest method, see [16] for more detail)

5.2 Ground-Truth Problems

For the ground-truth problems, methods did not undergo hyperparameter tuning. Instead, each method was given the most frequently selected hyperparameter set from the black-box problems. Then, each method was allowed up to 1,000,000 evaluations or 8 hours of training time per dataset, whichever came first. Various levels of noise were also added to the datasets before training.

5.2.1 Results. Figure 6 shows the median solution rate of each SR method sorted in descending order. Bingo is shown to have a very high solution rate with no target noise compared to the other tested methods. However, Bingo's performance significantly worsens as increasing levels of target noise are added. Overall, Bingo ranks similarly to other methods that perform well on the black-box problems but worse on the ground-truth problems (*e.g.*, Operon [3] and GP-GOMEA [28]).

6 CONCLUSION

We introduce a customizable and performant framework for symbolic regression with genetic programming: Bingo. Bingo is shown to be competitive to other symbolic regression methods in tasks such as general regression and recovery of ground-truth equations. Furthermore, as shown in Figure 5, relative to the other methods tested in the benchmark, Bingo produces simple yet accurate models.

REFERENCES

[1] Geoffrey F. Bomarito, Patrick E. Leser, Nolan M. Strauss, Karl M. Garbrecht, and Jacob D. Hochhalter. 2022. Bayesian Model Selection for Reducing Bloat and Overfitting in Genetic Programming for Symbolic Regression. In *Proceedings of the 24th annual conference on Genetic and evolutionary computation (GECCO '22)*. Association for Computing Machinery, Boston, MA, USA.

[2] Geoffrey F. Bomarito, Tyler S. Townsend, Kristen M. Stewart, Kathryn V. Esham, John M. Emery, and Jacob D. Hochhalter. 2021. Development of interpretable, data-driven plasticity models with symbolic regression. 252 (2021), 106557.

[3] Bogdan Burlacu, Gabriel Kronberger, and Michael Kommenda. 2020. Operon C++: an efficient genetic programming framework for symbolic regression. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*. Association for Computing Machinery, New York, NY, USA, 1562–1570. https://doi.org/10.1145/3377929.3398099

[4] Chetan Chudasama, S. M. Shah, and Mahesh Panchal. 2011. Comparison of parents selection methods of genetic algorithm for TSP. In *International Conference on Computer Communication and Networks CSI-COMNET-2011, Proceedings*. Citeseer, 85–87.

[5] Joel S. Cohen. 2003. *Computer algebra and symbolic computation: Mathematical methods*. AK Peters/CRC Press.

[6] Lisandro Dalcín, Rodrigo Paz, and Mario Storti. 2005. MPI for Python. *J. Parallel and Distrib. Comput.* 65, 9 (Sept. 2005), 1108–1115. https://doi.org/10.1016/j.jpdc.2005.03.010

[7] Vinicius Veloso De Melo, Benjamin Fowler, and Wolfgang Banzhaf. 2015. Evaluating methods for constant optimization of symbolic regression benchmark problems. In *2015 Brazilian conference on intelligent systems (BRACIS)*. IEEE, 25–30.

[8] François-Michel De Rainville, Félix-Antoine Fortin, Marc-André Gardner, Marc Parizeau, and Christian Gagné. 2014. DEAP: enabling nimbler evolutions. *ACM SIGEVOlution* 6, 2 (2014), 17–26.

[9] Z-Flores Emigdio, Leonardo Trujillo, Oliver Schütze, Pierrick Legrand, et al. 2014. Evaluating the effects of local search in genetic programming. In *EVOLVE-A Bridge between Probability, Set Oriented Numerics, and Evolutionary Computation V*. Springer, 213–228.

[10] Daniel JA Hills, Adrian M Grütter, and Jonathan J Hudson. 2015. An algorithm for discovering Lagrangians automatically from data. *PeerJ Computer Science* 1 (2015), e31.

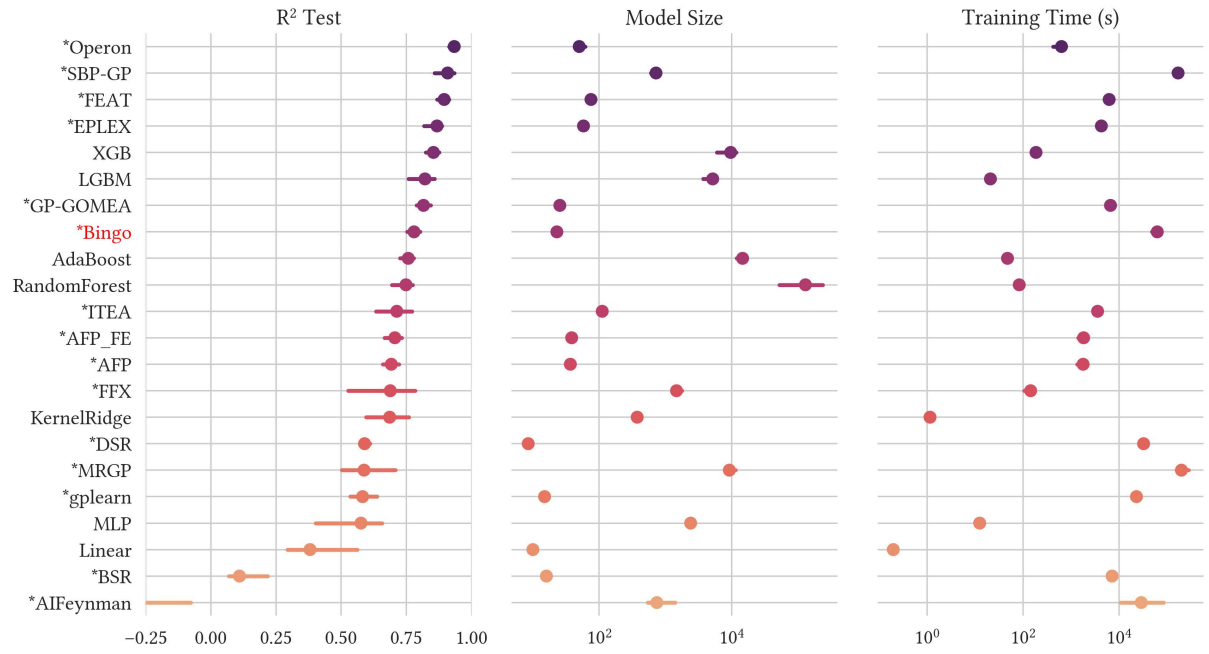

Figure 4: Black-box problem results.

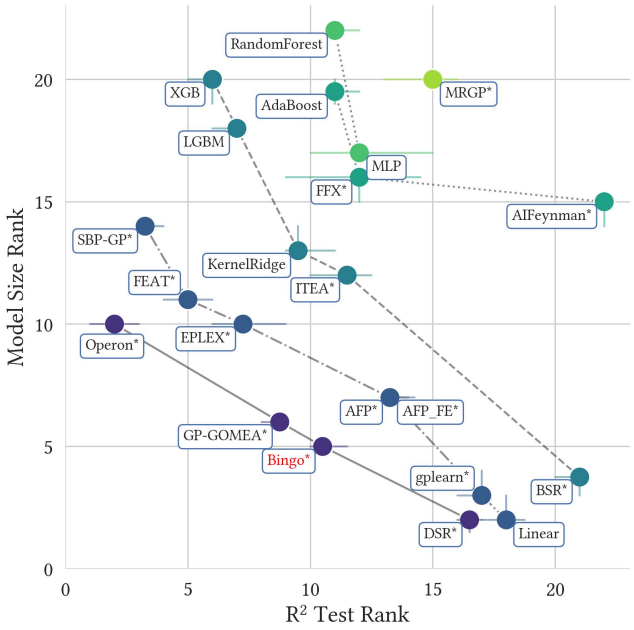

Figure 5: Black-box problem median R^2 rank vs. median model size rank.

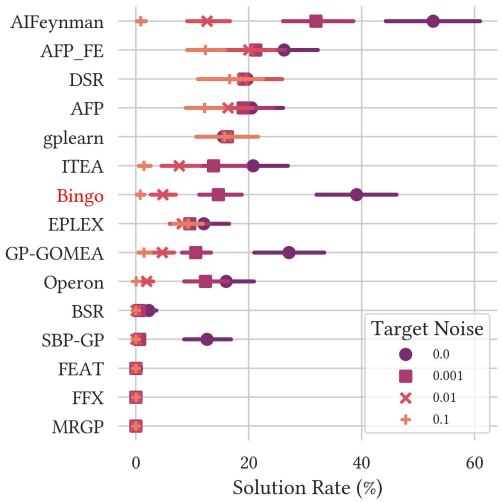

Figure 6: Ground-truth problem solution rates per method.

[11] Wenzel Jakob, Jason Rhinelander, and Dean Moldovan. 2016. pybind11 — Seamless operability between C++11 and Python. https://github.com/pybind/pybind11

[12] Khalid Jebari and Mohammed Madiafi. 2013. Selection methods for genetic algorithms. *International Journal of Emerging Sciences* 3, 4 (2013), 333–344.

[13] George Em Karniadakis, Ioannis G Kevrekidis, Lu Lu, Paris Perdikaris, Sifan Wang, and Liu Yang. 2021. Physics-informed machine learning. *Nature Reviews Physics* 3, 6 (2021), 422–440.

[14] Michael Kommenda, Gabriel Kronberger, Stephan Winkler, Michael Affenzeller, and Stefan Wagner. 2013. Effects of constant optimization by nonlinear least squares minimization in symbolic regression. In *Proceedings of the 15th annual conference companion on Genetic and evolutionary computation*. 1121–1128.

[15] John R. Koza. 1992. *Genetic programming: on the programming of computers by means of natural selection*. MIT Press, Cambridge, Mass.

[16] William La Cava, Patryk Orzechowski, Bogdan Burlacu, Fabrício Olivetti de França, Marco Virgolin, Ying Jin, Michael Kommenda, and Jason H. Moore. 2021. Contemporary Symbolic Regression Methods and their Relative Performance. *arXiv:2107.14351 [cs]* (July 2021). http://arxiv.org/abs/2107.14351 arXiv:

2107.14351.
[17] William La Cava, Tilak Raj Singh, James Taggart, Srinivas Suri, and Jason H. Moore. 2018. Learning concise representations for regression by evolving networks of trees. (2018). https://doi.org/10.48550/ARXIV.1807.00981
[18] Samir W. Mahfoud. 1995. *Niching Methods for Genetic Algorithms.* Ph. D. Dissertation. University of Illinois at Urbana-Champaign.
[19] W. N. Martin, Jens Lienig, and James P. Cohoon. 1997. Island (migration) models: evolutionary algorithms based on punctuated equilibria. *Handbook of Evolutionary Computation* 6 (1997), 101–124.
[20] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf
[21] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.
[22] Brenden K. Petersen, Mikel Landajuela Larma, T. Nathan Mundhenk, Claudio P. Santiago, Soo K. Kim, and Joanne T. Kim. 2021. Deep symbolic regression: Recovering mathematical expressions from data via risk-seeking policy gradients. *arXiv:1912.04871 [cs, stat]* (April 2021). http://arxiv.org/abs/1912.04871 arXiv: 1912.04871.
[23] Michael Schmidt and Hod Lipson. 2007. Comparison of tree and graph encodings as function of problem complexity. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*. 1674–1679.
[24] Michael Schmidt and Hod Lipson. 2009. Distilling free-form natural laws from experimental data. *science* 324, 5923 (2009), 81–85.
[25] Michael Schmidt and Hod Lipson. 2010. Symbolic regression of implicit equations. In *Genetic Programming Theory and Practice VII*. Springer, 73–85.
[26] Michael D. Schmidt and Hod Lipson. 2008. Coevolution of Fitness Predictors. *IEEE Transactions on Evolutionary Computation* 12, 6 (Dec. 2008), 736–749. https://doi.org/10.1109/TEVC.2008.919006
[27] Michael D. Schmidt and Hod Lipson. 2010. Age-fitness pareto optimization. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation (GECCO '10)*. Association for Computing Machinery, New York, NY, USA, 543–544. https://doi.org/10.1145/1830483.1830584
[28] 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*. ACM, Berlin Germany, 1041–1048. https://doi.org/10.1145/3071178.3071287
[29] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, et al. 2020. SciPy 1.0: fundamental algorithms for scientific computing in Python. *Nature Methods* 17, 3 (March 2020), 261–272. https://doi.org/10.1038/s41592-019-0686-2

A CODE EXAMPLE

```python
import numpy as np  # for generating data
from bingo.evolutionary_algorithms.age_fitness import AgeFitnessEA  # evolutionary algorithm
from bingo.evolutionary_optimizers.island import Island  # evolutionary optimizer
# components for evaluation
from bingo.evaluation.evaluation import Evaluation
from bingo.symbolic_regression import ExplicitRegression, ExplicitTrainingData  # fitness function
from bingo.local_optimizers.continuous_local_opt import ContinuousLocalOptimization  # CLO
# generators
from bingo.symbolic_regression import ComponentGenerator, AGraphGenerator
# crossover and mutation
from bingo.symbolic_regression import AGraphCrossover, AGraphMutation

# hyperparams
POP_SIZE = 100
STACK_SIZE = 20

# make some data
x = np.linspace(-10, 10, 100).reshape([-1, 1])
y = x**2 + 3.5*x**3
training_data = ExplicitTrainingData(x, y)

# setup generator to generate equations using variables, constants, +, -, and *
component_generator = ComponentGenerator(x.shape[1])
component_generator.add_operator("+")
component_generator.add_operator("-")
component_generator.add_operator("*")

# agraphs of equations will have STACK_SIZE (20) commands and will use algebraic simplification
agraph_generator = AGraphGenerator(STACK_SIZE, component_generator, use_simplification=True)

# crossover and mutation
crossover = AGraphCrossover()
mutation = AGraphMutation(component_generator)

# defining evaluation
fitness = ExplicitRegression(training_data=training_data, metric="mse")
local_opt_fitness = ContinuousLocalOptimization(fitness, algorithm="lm")
evaluator = Evaluation(local_opt_fitness)

# setup evolutionary algorithm
ea = AgeFitnessEA(evaluator, agraph_generator, crossover, mutation, 0.4, 0.4, POP_SIZE)

# setup evolutionary optimizer
island = Island(ea, agraph_generator, POP_SIZE)

# run!
opt_result = island.evolve_until_convergence(max_generations=500, fitness_threshold=1.0e-4)

# print best individual from last generation
print(opt_result.get_best_individual())
```