Taylor Genetic Programming for Symbolic Regression

Anonymous Author(s)

ABSTRACT

Genetic programming (GP) is a commonly used approach to solve symbolic regression (SR) problems. Compared with the machine learning or deep learning methods that depend on the pre-defined model and the training dataset for solving SR problems, GP is more focused on finding the solution in a search space. Although GP has good performance on large-scale benchmarks, it randomly transforms individuals to search results without taking advantage of the characteristics of the dataset. So, the search process of GP is usually slow, and the final results could be unstable. To guide GP by these characteristics, we propose a new method for SR, called Taylor genetic programming (TaylorGP). TaylorGP leverages a Taylor polynomial to approximate the symbolic equation that fits the dataset. It also utilizes the Taylor polynomial to extract the features of the symbolic equation: low order polynomial discrimination, variable separability, boundary, monotonic, and parity. GP is enhanced by these Taylor polynomial techniques. Experiments are conducted on three kinds of benchmarks: classical SR, machine learning, and physics. The experimental results show that TaylorGP not only has higher accuracy than the nine baseline methods, but also is faster in finding stable results.

CCS CONCEPTS

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

KEYWORDS

Taylor polynomials, genetic programming, symbolic regression

ACM Reference Format:

Anonymous Author(s). 2022. Taylor Genetic Programming for Symbolic Regression. In *Proceedings of The Genetic and Evolutionary Computation Conference 2022 (GECCO '22)*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Symbolic regression (**SR**) refers to finding a symbolic equation f_θ fitted to a given dataset (X, Y) from the mathematical expression space, i.e., $f_\theta(X) = Y$. The mathematical expression space is huge, even for rather simple symbolic equations. For example, if a symbolic equation is represented by a binary tree with a maximum depth of 4, with 20 variables $(x_1, x_2, ..., x_{20})$ and 18 basic functions (such as +, − and sqrt), the size of the space is 8.2×10^{162} [26]. Therefore, finding a good symbolic equation for the data from the huge possible search space is a challenging task.

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

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Research communities of evolutionary computation (**EC**) and machine learning (**ML**) have been trying to solve the SR problems from their perspectives. The EC methods, especially genetic programming (**GP**) methods [14, 27, 39, 48], are designed to search the mathematical expression space by evolving the encoding of each individual in a population. The main advantage of the GP approach is that the algorithm components are generalizable and adaptive, including processes such as selection, crossover, mutation, and fitness evaluation. Using these components, the GP algorithm randomly searches for a model f_θ that fits the dataset in the mathematical expression space, unlike machine learning methods (e.g., neural networks) that try to find and optimize a set of parameters θ under known models f. However, the GP search process usually does not consider the features of the given dataset, which could be an opportunity for improvement. The ML (including neural networks) methods [2, 23, 31, 36, 36, 46, 51] find parameters θ in pre-defined models f so that the models fit the dataset, i.e., $f_\theta(X) = Y$. Machine learning methods heavily utilize the features of a dataset to guide the search and optimization for parameters. Therefore, the parameter search process is effective. However, when solving SR problems, though the machine learning approaches could quickly recover some correct symbolic equations f_θs , the results are commonly biased by the pre-defined regression model fs as well as the training dataset. For example, in a recent evaluation [5] of algorithms for SR problems using large-scale benchmarks, the results show that the top three approaches are still GP-based approaches, and GP approaches still have a substantial advantage over machine learning-based approaches.

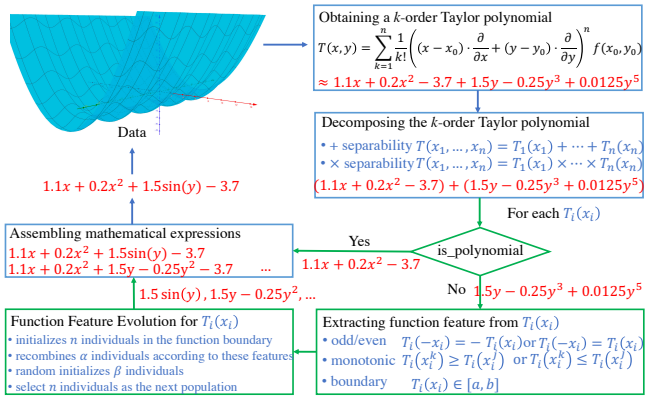


Figure 1: Taylor genetic programming.

In this paper, we propose a new GP approach called the Taylor Genetic Programming (TaylorGP). To overcome the common GP's drawback – a search process without considering the data features, we embed Taylor features into GP and leverage the features to guide GP to search for solutions. TaylorGP, as shown in Figure 1, first obtains a k-order Taylor polynomial at a point from the

given dataset. According to the Taylor's theorem, the k-order Taylor polynomial approximates a smooth function that fits the given dataset near the point. Moreover, the k-order Taylor polynomial can show the local features of the smooth function (called **Taylor features**). Taylor features include three key components: variable separability, low order polynomial discrimination, and function feature. Using the variable separability, TaylorGP can decompose the multivariate Taylor polynomial T into multiple univariate Taylor polynomials $T_i s$. For example, the two-variable Taylor polynomial "$1.1x + 0.2x^2 - 3.7 + 1.5y - 0.25y^3 + 0.0125y^5$" can be decomposed into two univariate Taylor polynomials "$1.1x + 0.2x^2 - 3.7$" and "$1.5y - 0.25y^3 + 0.0125y^5$" according the variable separability. TaylorGP then applies the polynomial discrimination to determine whether each T_i is a polynomial. If T_i is not a polynomial, TaylorGP runs the function feature evolution method to find symbolic equations to fit T_i. The function feature evolution method creates a set of rules based on the function features to recombine λ individuals. The method also employs the individual initialization method to randomly generate β individuals to prevent premature convergence. TaylorGP finally assembles mathematical expressions of all $T_i s$ to generate the final symbolic equation.

As the Taylor features are directly induced from the coefficients in the k-order Taylor polynomial, obtaining the features are simple and quick. For example, the coefficient of each two-variable product term "$x^m y^n$" in "$1.1x + 0.2x^2 - 3.7 + 1.5y - 0.25y^3 + 0.0125y^5$" is zero, meaning the Taylor polynomial is variable separable. So, embedding these features into GP does not increase the GP's computing time complexity. Moreover, the search process guided by the Taylor features enables TaylorGP to find a correct symbolic equation quicker than without using the features.

The main contributions of this paper are the following:

(1) We propose a simple yet powerful Taylor genetic programming (TaylorGP) method for symbolic regression. TaylorGP combines the general characteristics of GP's solution search strategies (e.g., mutation, crossover) with the ML (NN)'s feature-directed search.

(2) We design a new Taylor feature extraction method. Using a Taylor polynomial obtained from a dataset, the method can map the dataset into features that can represent the properties of a target symbolic equation. Moreover, we create a function feature evolution method to transform individuals according to these features.

(3) We demonstrate that TaylorGP significantly outperforms state-of-the-art approaches, such as FFX[36], GSGP [41], BSR [19], SVM, and XGBoost, on the three types of benchmarks: classical SR [37], AIFeynman [51], and Penn machine learning benchmarks [44].

The remainder of this paper is organized as follows. In Section 2, we detail related work. Section 3 presents the Taylor features. Then, we propose the Taylor genetic programming in Section 4. Section 5 and 6 report the experimental results and discussion. Finally, we conclude the paper in Section 6.

2 RELATED WORK

2.1 Machine learning for symbolic regression

The regression analysis in machine learning, such as linear regression [55], SVM [10], XGBoost [8], and neural network(NN) [33], can be viewed as a special case of SR. Different from SR that needs to find both the model and its parameter values, machine learning aims to find the values (θ) of parameters in a pre-defined model f, so that $f_\theta(X) = Y$. Many ML methods, such as deep neural networks, usually applies the gradient descent method to obtain the target parameter set θ. Neural networks [2, 24, 31, 35, 46, 47] are trained to learn from a dataset (X, Y) to generate a mathematical equation $f(X) = Y$ according to the features of the training dataset $\{(X, Y), \mathcal{F}\}$. For example, GrammarVAE (GVAE) [31] trains a variational autoencoder [24] to directly encode from and decode to the parse trees using a context-free grammar. Recently, DSR [46] employed a recurrent neural network trained by a reinforcement learning algorithm (RL) for SR. The algorithm uses a risk-seeking policy gradient to emit a distribution over tractable mathematical expressions. According to the distribution, DSR samples the mathematical expressions with constant placeholders and obtains these constants with the nonlinear optimization algorithm – BFGS [15]. Inspired by the recent successes of pre-trained models on large datasets, such as BERT [13] and GPT [4], NeSymReS [2] pre-trains a model called Set Transformer [34] on hundreds of millions of equations to generate a distribution of mathematical expressions according to the given dataset (X, Y). In addition, NeSymReS samples mathematical expressions with constant placeholders by the beam search on the distributions and uses BFGS to optimize these constants.

Besides the above neural networks for SR, EQL [35, 47] designs a shallow fully-connected neural network where standard activation functions (e.g., $tanh$, ReLU) are replaced with basic functions (e.g., "+", "×, "sin). Once the neural network is trained, it can represent a symbolic equation fitted to the given dataset. AIFeynman [50, 51] employs neural networks to map the given dataset into simplifying properties (e.g., symmetries, separability, and compositionality). The method then uses a brute-force recursive algorithm guided by these simplifying properties and a suite of physics-inspired techniques to search possible symbolic expressions.

The above machine learning methods, especially deep learning methods, have an excellent ability to discover mathematical equations on some specific benchmarks. However, experiments on the large-scale benchmarks [5] indicate that the mathematical equations found by ML or DL are less accurate than those found by GP-based methods. For example, four of the top five methods and six of the top ten methods are GP-based methods, and the other top methods are ensemble tree-based methods, such as XGBoost and LightGBM [20]. Surprisingly, the top methods do not include the two neural network methods, DSR and AIFeynman. Furthermore, the neural network methods seem to be more dependent on the training dataset. Neural networks could not discover a correct mathematical equation if their structures (layer by layer) could not extract valid features from the dataset.

2.2 Genetic programming for symbolic regression

GP [27] is still a commonly used approach to deal with SR. GP uses evolutionary operators[16] – crossover, mutation, and selection, to change the individual encoding and generate better offspring for

searching a solution in the mathematical expression space. Various GPs employ different individual encodings to represent mathematical equations, such as tree-encoded GPs [7, 27, 38, 41, 49], graph-encoded GPs [40, 48], and linearly encoded GPs [3, 14].

For the mathematical expression space, the presence of real constants accounts for a significant portion of the size of the space. For example, the size of the aforementioned problem in the Introduction section is 8.2×10^{162}. In comparison, without real constants, its size is 1.054×10^{19} [26]. So, some GP methods [25, 28, 56] with a constant optimizer are proposed to search the space. These methods represent the skeleton of a mathematical expression by using constant placeholders. And a constant optimizer is used to find the values in these constant placeholders. AEG-GP [25, 26] uses the abstract expression grammar to represent the skeleton of a mathematical expression and utilizes PSO [22] to find constant values. Unlike AEG-GP, PGE [56] is a deterministic SR algorithm that offers reliable and reproducible results. While PGE maintains a tree-based representation and Pareto non-dominated sorting from GP, it replaces the genetic operators and random numbers with grammar rules. The method also uses nonlinear regression to fit the constants of a mathematical equation. The approaches to GP-based feature engineering, such as GP-based feature construction [28], MRGP [1], FEW [32], M3GP [43], and FEAT [6], utilize EC to search for possible representations and couple with an ML model to handle the parameters of the representations. Different from GP-based feature engineering approaches, FFX [36] is a deterministic SR algorithm. It enumerates a massive set of basic features (basis functions–$B_i(x)$) by a production rule. It then find coefficient values(a) in "$y = a_0 + \sum_{i=1}^{N} a_i \times B_i(x)$" by using pathwise regularized learning.

All the above GP methods search the mathematical expression space without considering the features of the given dataset. It means that they construct or change an individual without using the information of the given dataset. While these GP-based feature engineering approaches evaluate each feature (basic function) by the given dataset, they still do not use the information from the dataset to generate operators to transform these basic functions. Unlike these above GPs, semantic-based GPs can utilize the dataset characteristics to transform and select individuals in the search space. For example, geometric semantic GP(GSGP) [41] creates the geometric crossover and mutation to generate an individual from its parent(s) in a way that guarantees to preserve their geometric properties. Following the idea of the geometric operators in GSGP, many geometric operators [7, 7, 30] and semantic backpropagation operators [29, 45, 53] are proposed to let GP execute the semantic search based on the given dataset.

The other research line uses the hybrid of a neural network and a GP, called **DL-GP**. DL-GPs [11, 42, 57, 58] leverage a neural network to obtain features from the given dataset and apply these features to guide GP. For example, Xing et al. [57] design an encoder-decoder neural network based on super-resolution ResNet to predicate the importance of each mathematical operator from the given data; and utilize the importance of each mathematical operator to guide GP. Cranmer et al. [11] train a graph neural network to represent sparse latent features of the given dataset, and employ GP to generate symbolic expressions fitted to these latent features. The DL-GPs still depend on the time-consuming training work and the training dataset.

Like AIFeynman, TaylorGP also needs to extract the symbolic equation's properties (e.g., separability and low-order polynomial) from the given dataset. However, the difference is that AIFeynman employs a neural network to obtain these properties while TaylorGP achieves the goal using the coefficients in the Taylor series on the given dataset. Therefore, TaylorGP does not need to train a model and does not depend on the training data. Compared with semantic-based GPs, TaylorGP does not need complex and time-consuming semantic operators to generate individuals since its operators are very simple.

3 TAYLOR FEATURES ANALYSIS

3.1 Obtaining a Taylor polynomial

Taylor's theorem [18] states that if a function f has $n+1$ continuous derivatives on an open interval containing a, for each x in the interval,

$$f(x) = \left[\sum_{k=0}^{n} \frac{f^{(k)}(a)}{k!} (x-a)^k \right] + R_{n+1}(x). \tag{1}$$

So, the k-order Taylor polynomial ($\sum_{k=0}^{n} \frac{f^{(k)}(a)}{k!}(x-a)^k$) approximates to f around a.

Given a dataset (X, Y), for any point $(x_0, y_0) \in (X, Y)$, the k-order Taylor polynomial around the point can be obtained by the following three steps. First, select k points ($\{(x_1, y_1), ..., (x_k, y_k)\}$) around (x_0, y_0) from the dataset. Next, according to the selected k points, gather k k-order Taylor polynomials by Equation 2.

$$\begin{cases} (x_1 - x_0) f'(x_0) + \ldots + \frac{(x_1 - x_0)^k}{k!} f^{(k)}(x_0) = f(x_1) - f(x_0) \\ (x_2 - x_0) f'(x_0) + \ldots + \frac{(x_2 - x_0)^k}{k!} f^{(k)}(x_0) = f(x_2) - f(x_0) \\ \cdots \\ (x_k - x_0) f'(x_0) + \ldots + \frac{(x_k - x_0)^k}{k!} f^{(k)}(x_0) = f(x_k) - f(x_0) \end{cases} \tag{2}$$

, where $f(x_i) = y_i$. The final step is to obtain the k derivatives (F) by Equation 3

$$F = DA^{-1} \tag{3}$$

, where $F = [f'(a), f''(a), ..., f^{(k)}(a)]^T$, $D = [y_1 - y_0, y_2 - y_0, ..., y_k - y_0]^T$. For each $a_{ij} \in A$, $a_{ij} = \frac{(x_i - x_0)^j}{j!}$. F can generate the k-order Taylor polynomial.

3.1.1 Scaling to a high dimensional dataset. Mathematically, the higher the order of k, the more accurate the Taylor polynomial is. However, in practice, k can not be too high in a high dimensional dataset D because of the two following limitations. According to the Taylor's theorem for multivariate functions [17], the n-variable k-order Taylor polynomial has C_{n+k}^{n} terms. So, $C_{n+k}^{n} - 1$ points need to be sampled around the point to obtain the Taylor polynomial. However, if k is too high, $C_{n+k}^{n} - 1$ will be greater than all points in D. This is impossible. Therefore, k must be limited so that $C_{n+k}^{n} - 1 < |D|$. The other limitation is that, if k is too high, the inverse of the matrix A in Equation 3 would be a big challenge to compute since A is an ultra-large-scale ($(C_{n+k}^{n} - 1) \times (C_{n+k}^{n} - 1)$) matrix. So, we usually set $k = 1$ *or* 2 in a high-dimensional dataset because of the two limitations.

3.2 Extracting Taylor features

Since the above k-order Taylor polynomial is generated from a given dataset (X, Y), it can approximate a function f that fits the dataset (i.e., $f(X) = Y$) around a point (x_0, y_0). It can also represent some f's local features, called Taylor features. This paper discusses the Taylor features, low order polynomial, variable separability, function boundary, monotony and parity.

3.2.1 Low order polynomial discrimination. For SR, a key problem is to discriminate whether there is (or only) a low-order polynomial that can represent the given dataset. If it exists, a linear regression algorithm can be used to get it and the algorithm could solve SR quickly. The k-order Taylor polynomial can easily solve the discrimination problem owing to its coefficients. If a function is a k-order polynomial, its Taylor expansion at a point is also a k-order polynomial. For example, for "$1.1x + 0.2x^2 - 3.7$", its Taylor expansion at $x = 1$ is also '$1.1x + 0.2x^2 - 3.7$'. While, if a function is not a k-order polynomial, its Taylor expansion at a point is an infinite-order polynomial. For $1.5sin(x)$, its Taylor expansion is an infinite order polynomial. So, for the k-order Taylor polynomial obtained from a dataset, in each term whose degree is greater than i ($i < k$), if the coefficient is zero, the function that k-order Taylor polynomial approximates is a low i-order polynomial.

3.2.2 Variable separability. For a multivariate function $f(x_1, ..., x_n)$, if there is an operator "$\circ$" that lets $f(x_1, ..., x_n) = f_1(x_i, ..., x_k) \circ f_2(x_m, ..., x_p)$ where the two variable sets, $\{x_i, ..., x_k\}$ and $\{x_m, ..., x_p\}$, both belong to $\{x_1, ..., x_n\}$, and $\{x_i, ..., x_k\} \cap \{x_m, ..., x_p\} = \phi$, it is called "$\circ$" separability. The separability property can decompose a complex multivariate function into multiple simple functions.

If "$\circ$" is addition or multiplication, it is called **addition separability** or **multiplication separability**, respectively. The n-variable k-order Taylor polynomial can represent the two separability properties, respectively. For the Taylor polynomial, if the coefficient in each multi-variable term is zero, the function that the Taylor polynomial approximates is addition separability. As shown in Figure 1, the Taylor expansion of "$1.1x + 0.2x^2 + 1.5sin(y) - 3.7$" is "$1.1x + 0.2x^2 - 3.7 + 1.5y - 0.25y^3 + 0.0125y^5$", where the coefficient in each multi-variable term is zero, i.e., c in each cx^iy^j is 0. So, according to the addition separability, the Taylor polynomial is decomposed into multiple polynomials, such as "$1.1x + 0.2x^2 + 1.5sin(y) - 3.7 \approx (1.1x + 0.2x^2 - 3.7) + (1.5y - 0.25y^3 + 0.0125y^5)$".

The above method about addition separability also can be used to discriminate multiplication separability. Because, if a function is multiplication separability, i.e., $f(x_1, ..., x_n) = f_1(x_i, ..., x_k) \times f_2(x_m, ..., x_p)$, then $\log f(x_1, ..., x_n) = \log f_1(x_i, ..., x_k) + \log f_2(x_m, ..., x_p)$. So, if the n-variable k-order Taylor polynomial obtained after computing the log of the dataset is addition separability, then the function is the multiplication separability.

3.2.3 Boundary. The k-order Taylor polynomial can be used to evaluate the boundary of the function f that it approximates to at the interval $[x_a, x_b]$. Since it is a polynomial, its boundary is computed by interval arithmetic [12]. For example, the boundary of the Taylor polynomial "$1.1x + 0.2x^2 - 3.7$", where $x \in [-1, 1]$, is "$[-1.1, 1.1] + 0.2 \times [0, 1] - 3.7 = [-4.8, -2.4]$".

3.2.4 Monotonic. For all points (x_i, y_i) in a dataset (X,Y), if $y_i \geq y_j$ and $x_i \geq x_j$, the function that the dataset represents is a monotonic-increasing function. Otherwise, if $y_i \geq y_j$ and $x_i \leq x_j$, the function is a monotonic-decreasing function.

3.2.5 Parity. If the k-order Taylor polynomial $T(x)$ is an odd or even function, i.e., $T(-x) = -T(x)$ or $T(-x) = T(x)$, then the function that it approximates is also an odd or even function. While the method is simple, testing all points is time-consuming.

Another method is to count odd-order terms and even-order terms in the k-order Taylor polynomial except for the 0-order term (constant term). If the k-order Taylor polynomial only contains odd (or even) order terms, it is an odd (or even) function. For example, given that $f(x) = 1.5sin(x) - 3.7$ and its Taylor polynomial is $1.5x - 0.25x^3 + 0.0125x^5 + ... + 4.217e - 15x^{17} - 3.7$ at the point $x = 0$, after removing the 0-order term "-3.7", the Taylor polynomial only contain odd-order terms, such as "$0.25x^3$"and "$0.0125x^5$". So, it is an odd function.

4 TAYLOR GENETIC PROGRAMMING

TaylorGP, as shown in Figure 1, includes the following six steps: 1) obtaining a Taylor polynomial T from a given dataset, 2) decomposing the Taylor polynomial into multiple simple Taylor polynomials ($\{T_1, T_2, ..., T_n\}$), 3) discriminating the low order polynomial, 4) extracting function features, 5) running the function feature evolution method, and 6) assembling mathematical expressions. How to execute the steps: 1), 2), 3), and 4) has been introduced in Section 3. Step 6) is simple, which only composes the mathematical expressions found by each simple Taylor polynomials into various complete mathematical equations and evaluates them. So, the following content details step 5).

The function feature evolution method (FFEM), as shown in Algorithm 1, evolves individuals based on the **function feature** F that includes boundary, monotonic, and parity. FFEM mainly contains two evolvable operators, individual initialization (initIndividualByFeatures) and individual recombination (recombineByFeatures). The individual initialization operator randomly generates individuals that satisfy the function feature. The individual recombination operator transforms individuals to ensure that the generated individuals satisfy the function feature. In each generation, FFEM leverages individual recombination to produce offspring with the probability α; utilizes the individual initialization to produce offspring with the probability β; saves individuals as other offspring with the probability $(1 - \alpha - \beta)$.

4.1 Individual initialization

The probability of randomly generating an individual that satisfies the function feature F is very small. And the process for obtaining N number of these candidate individuals is very time-consuming. To speed up the process, the individual initialization operator first segments the mathematical expression space into many sub-spaces. It then evaluates the function features of these sub-spaces. Finally, it randomly selects the sub-spaces that satisfy F and randomly generates individuals in these sub-spaces until they satisfy F.

4.1.1 Segmenting mathematical expression space . A tree can be used to represent a mathematical expression. A tree with depth h

Algorithm 1 function feature evolution method

Input: (X_i, Y_i), α, β, *threshold*, *maxGen*,*F*
Output: *best*

```
P ← initIndividualByFeatures(F,popsize=N)
best ← selectBestIndividual(P)
while best.fitness ≤ threshold and g < maxGen do
    for all i = 1 to N do
        p1, p2 ← randomSelectTwoIndividuals(P)
        if rand() < α then
            child ← recombineByFeatures(p1, p2,F)
        else if rand() < α + β then
            child ← initIndividualByFeatures(F,popSize=1)
        else
            child ← p1
        end if
        nextP[i] ← child
    end for
    P ← nextP
    best ← min(best, selectBestIndividual(P,(Xi, Yi)))
    g++
end while
return best
```

also shows a sub-space that contains all mathematical expressions expanded from the tree. Moreover, all trees with h represent a segment of the mathematical expression space. For example, given a basic function set $\{+, sin\}$ and a variable set $\{x, c\}$, the mathematical expression space is divided into the sub-spaces encoded by trees with depth 3, such as "++*sincxx*", "+++*xcc*", "+*sinsinxx*", and "*sin*+*xc*". The sub-space "++*sincxx*" is a mathematical expression "$c + x + sin(x)$", which contains all mathematical expressions expanded from "++*sincxx*", such as "$++sin+cxxx$"="$(c + x) + sin(x) + x$".

4.1.2 Evaluating sub-space. Interval arithmetic [12] can be used to compute the boundary of a sub-space. In the tree that represents the sub-space, if there is a path from a leaf node to the root node consisting of unbound functions $(x, +, -, \times, /, a^x, \ln, ...)$, the boundary of the sub-space is $[-\infty, \infty]$. Otherwise, it is computed by interval arithmetic. For example, for the sub-space "++*sincxx*", there is a path "$x++$", so its boundary is $[-\infty, \infty]$. There is no such path for the sub-space "+*sinsinxx*", so its boundary is [-2,2] obtained by interval arithmetic.

A sub-space's non-monotonic or monotonic increasing/decrease is determined by its derivative d. If $d \geq 0$ (or $d \leq 0$) in all variable values, it is a monotone increasing (decreasing) function. Otherwise, it is a non-monotone function. The sub-space is an odd/even function according to "$f(-x) = -f(x)$" or "$f(-x) = f(x)$" for all values in x. For the sub-space "++*sincxx*=$c + x + sin(x)$", its derivative is "$1 + cos(x)$", meaning that it is a monotone-increasing function. Owing to "c" in "$c+x+sin(x)$", it is a non-odd and non-even function.

4.1.3 Generating individual. The method "generating individual" obtains the segmented sub-spaces whose boundaries contain the given boundary. It then randomly selects a sub-space from these sub-spaces. If the sub-space does not satisfy the given monotony and parity requirements, the method randomly generates a new individual from the sub-space until it satisfies the given function features. Otherwise, the method randomly generates an individual based on the following rules, as listed in Table 1. For example, given the function features $\{[-10, 10], O\}$, for the selected sub-space "++*sincxx*=$c + x + sin(x)$", it is a non-odd and non-even function. So, the method randomly generates individuals until an individual is an odd function whose boundary contains $[-10, 10]$. For the selected sub-space "++*sinxxx*=$2x + sin(x)$", it is an odd function. So, the method randomly constructs an odd function (e.g., $sin(x)$). It then combines the odd function and the selected sub-space according to an operator randomly selected (e.g. "+") from $\{+, -, \times, /, f(g(x)\}$. With these steps the method finally generates an individual "$2x + 2sin(x)$" whose boundary contains $[-10, 10]$.

Table 1: Function Combination Rules

op	$f(x)$	$g(x)$	results
+	odd	odd	odd
+	even	even	even
−	odd	odd	odd
−	even	even	even
×, /	odd	odd	even
×, /	even	even	even
×, /	odd	even	odd
$f(g(x))$	even	even	even
$f(g(x))$	odd	odd	odd
$f(g(x))$	even	odd	even
$f(g(x))$	odd	even	even
+	↗	↗	↗
×	↗	↗	↗
$f(g(x))$	↗	↗	↗

"↗" represents a monotone-increasing function. The monotone-decreasing function has similar properties to the monotone-increasing function.

4.2 Individual recombination

According to the rules in Table 1, the individual recombination operator recombines two individuals from the population to construct an individual that satisfies the given function feature. Meanwhile, if the recombined individual exceeds the limit length, the operator prunes it to avoid the individual bloating. For example, given a function feature – odd function, the operator recombines the two individuals "$2x + 2sin(x)$" and $x + x^3$ with "+". The recombined individual "$2x + 2sin(x) + x + x^3$" exceeds the limited length 12. It then prunes "$sin(x)$" in the individual and replaces "$sin(x)$" with "x" according to "$f(g(x))$". The method in the end generates the individual "$5x + x^3$" whose length is 9.

5 EXPERIMENT

5.1 Datasets

We evaluate the performance of TaylorGP on three kinds of benchmarks: classical Symbolic Regression Benchmarks (**SRB**) [37], Penn Machine Learning Benchmarks (**PMLB**) [44], and Feynman Symbolic Regression Benchmarks (**FSRB**) [51]. SRB consists of twenty-three SR problems derived from the five canonical symbolic regression benchmarks, Nguyen [52], Korns [25], Koza [9], Keijzer

[21], and Vladislavleva [54]. PMLB includes seven regression tasks and three classification tasks. FSRB contains forty-eight Feynman equations in [51]. The distribution of the total 81 benchmark sizes by samples and features is shown in Figure 2. The details of these benchmarks are listed in the appendix.

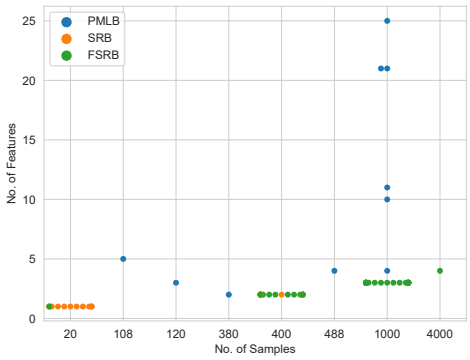


Figure 2: Properties of Benchmarks.

5.2 Algorithm Parameter Settings

We compare TaylorGP with two kinds of baseline algorithms [1]: four symbolic regression methods and five machine learning methods. The symbolic regression methods include **GPlearn**[2], **FFX** [36], geometric semantic genetic programming (**GSGP**)[41] and bayesian symbolic regression (**BSR**) [19]. The machine learning methods include linear regression (**LR**), kernel ridge regression (**KR**), random forest regression (**RF**), support vector machines (**SVM**), and **XGBoost** [8]. The detailed parameters of each algorithm are tuned according to Table 2.

6 RESULTS AND DISCUSSION

6.1 Performance Metrics

TaylorGP and nine baseline algorithms run 30 times on each benchmark. Their fitness results are listed in the appendix. In addition, the following R^2 test [5] is introduced to evaluate the performance of these algorithms on these benchmarks.

$$\mathrm{R}^2 = 1 - \frac{\sum_{i=1}^{n} (\hat{y}_i - y_i)^2}{\sum_{i=1}^{n} (\bar{y} - y_i)^2}, \tag{4}$$

, where y_i is the value in the dataset, $\bar{y}$ is mean and $\hat{y}_i$ is the output value of the best solution.

Figure 3 illustrates the normalized R^2 scores of the ten algorithms running 30 times on all benchmarks. Since the normalized R^2 closer to 1 indicates better results, overall TaylorGP can find more accurate results than other algorithms. Moreover, TaylorGP's results are more stable. The normalized R^2 scores of the ten algorithms on each benchmark (in the appendix) show that TaylorGP can outperform the nine baseline algorithms on most benchmarks.

Table 3 shows that TaylorGP still outperforms the nine baseline algorithms on the pairwise statistical comparisons with the Wilcoxon signed-rank test. Except for TaylorGP, it is not easy to find one algorithm outperform all other algorithms consistently on the benchmarks.

[1]The nine baseline algorithms are implemented in SRBench [5]
[2]https://github.com/trevorstephens/gplearn

Table 2: Algorithm parameters

Name	Parameter	Value
TaylorGP	Function Set	+,−,×,÷,sin,cos,$ln(\|n\|)$,exp,$sqrt$
	Max Generations	10000
	Population Size	1000
	Crossover Rate	0.7
	Mutation Rate	0.2
	Copy Rate	0.1
	Stopping Threshold	1e-5
GPLearn	Function Set	+,−,×,÷,sin,cos,$ln(\|n\|)$,exp,$sqrt$
	max generations	10000
	Population Size	1000
	Crossover Rate	0.7
	Mutation Rate	0.2
	Copy Rate	0.1
	Stopping Threshold	1e-5
GSGP	Function Set	+,−,×,÷
	Max Generations	10000
	Population Size	1000
	Crossover Rate	0.7
	Mutation Rate	0.2
	Stopping Threshold	1e-5
BSR	Function Set	+,−,×,÷,sin,cos,$ln(\|n\|)$,exp,$sqrt$
	MM	10000
	k	2
	Stopping Threshold	1e-5
FFX	None	
LR	Normalize	FALSE
KR	Kernal	'linear', 'poly', 'rbf', 'sigmoid'
	Gamma	0.01,0.1,1,10
	Regularization	0.001,0.1,1
RF	Number of Estimators	10, 100, 1000
	Max Features	'sqrt','log2',None
SVM	Kernal	'linear', 'poly', 'rbf', 'sigmoid', 'precomputed'
XGBoost	Learning Rate	0.0001,0.01, 0.05, 0.1, 0.2
	Gamma	0,0.1,0.2,0.3,0.4

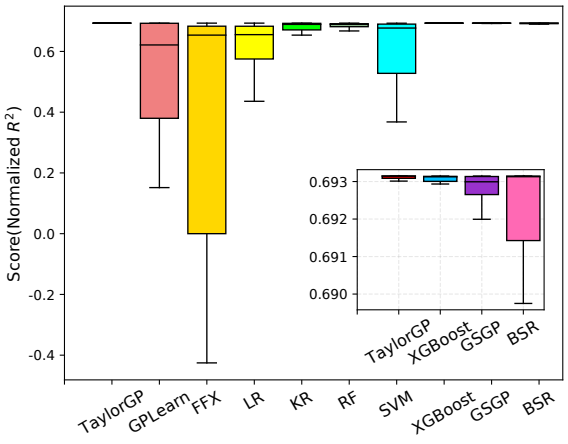


Figure 3: Normalized R^2 scores of the ten algorithms.

6.2 Discussion

Why does TaylorGP outperform the nine baseline algorithms on most benchmarks? The main reason is that the Taylor features can

Table 3: Wilcoxon signed-rank test of normalized R^2 scores for pairwise statistical comparisons.

	TaylorGP	GPLearn	FFX	LR	KR	RF	SVM	XGBoost	GSGP
GPLearn	**5.73e-13**								
FFX	**8.87e-11**	**6.55e-03**							
LR	**5.86e-15**	8.70e-01	9.82e-01						
KR	**2.73e-15**	1.00e+00	1.00e+00	1.00e+00					
RF	**4.31e-14**	1.00e+00	1.00e+00	1.00e+00	8.04e-01				
SVM	**2.84e-15**	5.34e-01	1.00e+00	7.74e-01	**2.28e-05**	**1.36e-05**			
XGBoost	**6.72e-05**	1.00e+00	1.00e+00	1.00e+00	1.00e+00	1.00e+00	1.00e+00		
GSGP	**1.08e-04**	1.00e+00	1.00e+00	1.00e+00	1.00e+00	1.00e+00	1.00e+00	**2.43e-03**	
BSR	**3.09e-03**	1.00e+00	1.00e+00	1.00e+00	1.00e+00	1.00e+00	1.00e+00	**3.62e-02**	9.84e-02

[1] bold number means that $p < 0.05$.

guide TaylorGP to search the problem space more effectively than the baselines. Compared with the other five ML methods, TaylorGP does not need to construct a predefined model to find a model that fits the given dataset. Therefore, on large-scale benchmarks, it can find better results. Compared with other GPs that need to search the whole mathematical expression space, TaylorGP's search space is smaller. So, it can find the correct results faster.

6.2.1 Convergence Analysis. We compare TaylorGP with the other three SR methods, GPLearn, GSGP, and BSR. Two benchmarks are used in the following evaluation. One benchmark is the "$x_0^{x_1}$" from SRB. The other is the "$U = \frac{1}{2}k_{spring}x^2$" from FSRB. We illustrate how the Taylor features help TaylorGP quickly find the correct results through the two benchmarks. Figures 4 and 5 show the processes of the four methods running on the two benchmarks, respectively.

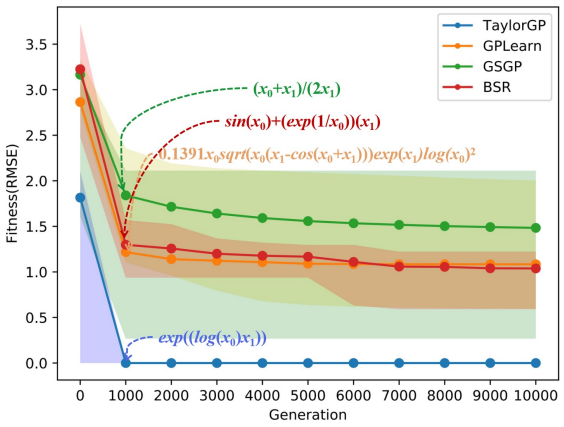


Figure 4: Convergence Comparison for "$x_0^{x_1}$".

For the benchmark "$x_0^{x_1}$" where x_0 and $x1$ both are in $[2, 4]$, compared with GPLearn, GSGP and BSR, TaylorGP can find the optimal result "$exp(log(x_0)x_1) = x_0^{x_1}$" at the 1000th generation. While the other three algorithms still can not find the optimal results until they run for 10,000 generations. TaylorGP first generates the Taylor polynomial "$(2.596e-3)x_0^4 + 2.932x_0^3x_1 - 7.828x_0^3 + 15.171x_0^2x_1^2 - 100.026x_0^2x_1 + 163.688x_0^2 + 11.398x_0x_1^3 - 167.424x_0x_1^2 + 710.31x_0x_1 - 932.144x_0 + 1.636x_1^4 - 47.859x_1^3 + 416.67x_1^2 - 1393.536x_1 + 1590.457$" at the point(2,2). According to the Taylor polynomial, the function boundary is $[4.233, 230.513]$, and the function is monotonically increasing. The two Taylor features can reduce the space used to initialize individuals. As "$\log(x_0)$", "$x_0 \times x_1$", and "$exp(x_0)$" all are monotone increasing functions at the range [2,4], they are very likely to be selected as initialized. The individual recombination that recursively merges the three functions using the operator $f(g(x))$ in Table 1 may generate "$exp(log(x_0)x_1)$". So, TaylorGP, compared with the other three algorithms, can initialize better individuals and get the optimal result earlier, as shown in Figure 4.

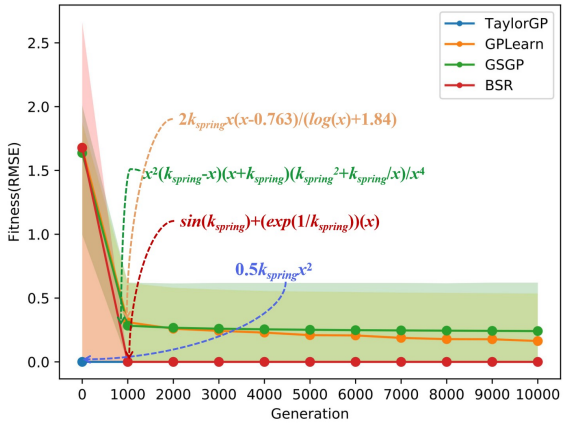
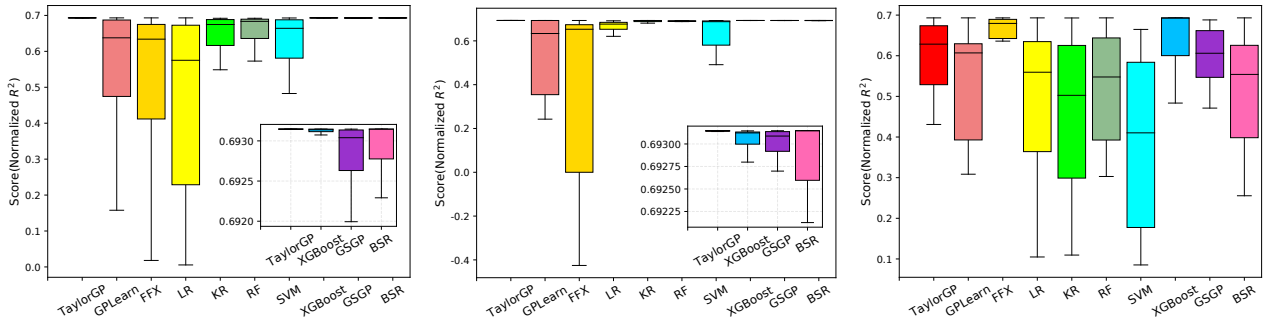


Figure 5: Convergence Comparison for "$U = \frac{1}{2}k_{spring}x^2$", where k_{spring} and x are variables.

For the Feynman benchmark "$U = \frac{1}{2}k_{spring}x^2$", TaylorGP finds the optimal results at the 0th generation, because TaylorGP can directly obtain $0.2k_{spring}x^2$ owing to the low polynomial discrimination. TaylorGP achieves the Taylor polynomial "$(1.454e-6)x_0^3 - (1.181e-6)x_0^2x_1 - (7.784e-6)x_0^2 + (0.5)x_0x_1^2 + (5.809e-6)x_0x_1 + (9.249e-6)x_0 - (2.53e-6)x_1^3 + (2.041e-5)x_1^2 - (6.481e-5)x_1 + (5.579e-5)$" from the benchmark. After omitting insignificant coefficients that are less than $e-4$, the Taylor polynomial is "$0.5x_0x_1^2$". As the RMSE of "$0.5x_0x_1^2$" is less than the stopping threshold $e-5$, it is the final result that TaylorGP finds.



(a) SRB (b) FSRB (c) PMLB

Figure 6: Normalized R^2 comparisons of the ten SR methods on SRB, FSRB and PMLB, respectively.

Besides the two figures, the figures of the convergence comparison that the four algorithms run on the other benchmarks are listed in the appendix.

6.2.2 Fitness Analysis. Figure 6 illustrates that TaylorGP, when compared with the nine baseline algorithms, can obtain more accurate and stable results on the two benchmarks, SRB and FSRB. However, on the benchmark PMLB, the two algorithms, FFX and XGBoost, outperform TaylorGP. This is due to PMLB has more features (variables) than the other two benchmark sets, Figure 6 shows that TaylorGP has the best performance (normalized R^2 score) on the-low dimensional datasets. In contrast, its performance degrades as the dataset's dimension increases. For a high dimension dataset, TaylorGP can only obtain a low order Taylor polynomial according to the analysis in Section 3.1.1. However, the low order Taylor polynomial may not approximate the real function that fits the given high-dimensional dataset. The Taylor features extracted from the Taylor polynomial may be incorrect or incomplete; therefore the features cannot help TaylorGP find a correct result.

6.2.3 The accuracy of extracting Taylor features. As the real function that fits the dataset in PMLB is unknown, Table 4 lists the accuracy of extracting each Taylor feature on SRB and FSRB (total 71 benchmarks). TaylorGP can correctly identify the two Taylor features, monotone and boundary, on all benchmarks, meaning that the two Taylor features always help Taylor reduce the search space.

However, TaylorGP recognizes the variable separability and even/odd function with low accuracies (12.5% and 36.7%). For identifying the variable separability and the odd/even function, TaylorGP requires that the Taylor polynomial can not contain some order terms, i.e., the coefficients in these order terms must be zero. However, as the Taylor polynomial approximates the real function around a point, some inconsistencies exist between the polynomial and the real function. The coefficients on these terms are slight errors. These slight error coefficients affect the recognition of the variable separability and the odd/even function. For example, for $sin(x)$, its Taylor polynomial at the point (0,0) is "$\frac{1}{1!}x-\frac{1}{3!}x^3+\frac{1}{5!}x^5-\frac{1}{7!}x^7+...$". However, according to Equation 3, TaylorGP sets a 4-order Taylor polynomial and obtains the polynomial "$0.015+\frac{1}{1!}x+0.003x^2-\frac{1}{3!}x^3$" from the given dataset. The polynomial is not an odd function due to the two coefficients, "0.015" and "0.003". To prevent this from happening, we set a threshold for these coefficients and omit the terms whose coefficients are less than the threshold. However, it is not easy to get a suitable threshold for the Taylor polynomial because of the diversity of datasets.

Table 4: The accuracy of extracting Taylor features on 71 benchmarks.

Taylor Features	Accuracy	Correct No	Ground Truth No
LowOrderPoly	73.9%	17	23
Separability	12.5%	3	24
Boundary	100.0%	71	71
Odd/even function	36.7%	18	49
Monotone	100.0 %	10	10

Although the two Taylor features (variable separability, odd/even function) have a low recognition accuracy, they still can help TaylorGP to find the correct symbolic equation, such as running TaylorGP on the two above benchmarks, "$x_0^{x_1}$" and "$U = \frac{1}{2}k_{spring}x^2$". So, TaylorGP can utilize the Taylor features to reduce its search space and speed up its search.

6.3 Conclusion

This paper proposes a new method called TaylorGP to search the mathematical expression space using Taylor features. As most of the Taylor features are obtained by the coefficients in a Taylor polynomial, the modeling process can be computationally efficient and straightforward to implement. TaylorGP leverages the two operators based on Taylor features, individual initialization, and individual recombination, to evolve the population. Experiments show that TaylorGP can quickly find the correct result with the help of the two evolution operators.

However, TaylorGP will degrade when the dataset dimension increases because of the local approximation of the Taylor polynomial. In a high-dimensional dataset, a low order Taylor polynomial obtained from the dataset only represents the dataset's local features, not global features. So, our future work will involve investigating how to utilize many low-order Taylor polynomials to represent global features in high-dimensional datasets.

REFERENCES

[1] Ignacio Arnaldo, Krzysztof Krawiec, and Una-May O'Reilly. 2014. Multiple regression genetic programming. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*. 879–886.

[2] Luca Biggio, Tommaso Bendinelli, Alexander Neitz, Aurelien Lucchi, and Giambattista Parascandolo. 2021. Neural Symbolic Regression that scales. In *Proceedings of the 38th International Conference on Machine Learning*, Vol. 139. PMLR, 936–945.

[3] Markus F Brameier and Wolfgang Banzhaf. 2007. *Linear genetic programming*. Springer Science & Business Media.

[4] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165* (2020).

[5] William La Cava, Patryk Orzechowski, Bogdan Burlacu, Fabricio Olivetti de Franca, Marco Virgolin, Ying Jin, Michael Kommenda, and Jason H. Moore. 2021. Contemporary Symbolic Regression Methods and their Relative Performance. In *Advances in Neural Information Processing Systems Datasets and Benchmarks Track*.

[6] William La Cava, Tilak Raj Singh, James Taggart, Srinivas Suri, and Jason Moore. 2019. Learning concise representations for regression by evolving networks of trees. In *International Conference on Learning Representations*.

[7] Q. Chen, B. Xue, and M. Zhang. 2019. Improving Generalization of Genetic Programming for Symbolic Regression With Angle-Driven Geometric Semantic Operators. *IEEE Transactions on Evolutionary Computation* 23, 3 (June 2019), 488–502.

[8] 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.

[9] Steffen Christensen and Franz Oppacher. 2002. An Analysis of Koza's Computational Effort Statistic for Genetic Programming. In *Genetic Programming*, James A. Foster, Evelyne Lutton, Julian Miller, Conor Ryan, and Andrea Tettamanzi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 182–191.

[10] Corinna Cortes and Vladimir Vapnik. 1995. Support-vector networks. *Machine learning* 20, 3 (1995), 273–297.

[11] Miles Cranmer, Alvaro Sanchez Gonzalez, Peter Battaglia, Rui Xu, Kyle Cranmer, David Spergel, and Shirley Ho. 2020. Discovering Symbolic Models from Deep Learning with Inductive Biases. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 17429–17442.

[12] Hend Dawood. 2011. *Theories of interval arithmetic: mathematical foundations and applications*. LAP Lambert Academic Publishing.

[13] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186.

[14] Candida Ferreira. 2001. Gene Expression Programming: a New Adaptive Algorithm for Solving Problems. *Complex Systems* 13, 2 (2001), 87–129.

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

[16] Stephanie Forrest. 1993. Genetic algorithms: principles of natural selection applied to computation. *Science* 261, 5123 (1993), 872–878.

[17] Mark H Holmes. 2009. Introduction to the foundations of applied mathematics. (2009).

[18] Harold Jeffreys, Bertha Jeffreys, and Bertha Swirles. 1999. *Methods of mathematical physics*. Cambridge university press.

[19] Ying Jin, Weilin Fu, Jian Kang, Jiadong Guo, and Jian Guo. 2019. Bayesian symbolic regression. *arXiv preprint arXiv:1910.08892* (2019).

[20] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. Lightgbm: A highly efficient gradient boosting decision tree. *Advances in neural information processing systems* 30 (2017), 3146–3154.

[21] Maarten Keijzer. 2003. Improving symbolic regression with interval arithmetic and linear scaling. In *European Conference on Genetic Programming*. Springer, 70–82.

[22] James Kennedy and Russell Eberhart. 1995. Particle swarm optimization. In *Proceedings of ICNN'95-international conference on neural networks*, Vol. 4. IEEE, 1942–1948.

[23] Samuel Kim, Peter Y Lu, Srijon Mukherjee, Michael Gilbert, Li Jing, Vladimir Čeperić, and Marin Soljačić. 2020. Integration of neural network-based symbolic regression in deep learning for scientific discovery. *IEEE Transactions on Neural Networks and Learning Systems* (2020).

[24] Diederik P Kingma and Max Welling. 2013. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114* (2013).

[25] Michael F Korns. 2011. Accuracy in symbolic regression. In *Genetic Programming Theory and Practice IX*. Springer, 129–151.

[26] Michael F Korns. 2013. A baseline symbolic regression algorithm. In *Genetic Programming Theory and Practice X*. Springer, 117–137.

[27] Koza and John R. 1994. Genetic Programming as a Means for Programming Computers by Natural Selection. *Statistics and Computing* 4, 2 (June 1994), 87–112. https://doi.org/10.1007/BF00175355

[28] Krzysztof Krawiec. 2002. Genetic programming-based construction of features for machine learning and knowledge discovery tasks. *Genetic Programming and Evolvable Machines* 3, 4 (2002), 329–343.

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

[30] Krzysztof Krawiec and Tomasz Pawlak. 2013. Locally geometric semantic crossover: a study on the roles of semantics and homology in recombination operators. *Genetic Programming and Evolvable Machines* 14, 1 (2013), 31–63.

[31] Matt J Kusner, Brooks Paige, and José Miguel Hernández-Lobato. 2017. Grammar variational autoencoder. In *International Conference on Machine Learning*. PMLR, 1945–1954.

[32] William La Cava and Jason Moore. 2017. A General Feature Engineering Wrapper for Machine Learning Using ϵ-Lexicase Survival. In *European Conference on Genetic Programming*. Springer, 80–95.

[33] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *nature* 521, 7553 (2015), 436–444.

[34] Juho Lee, Yoonho Lee, Jungtaek Kim, Adam Kosiorek, Seungjin Choi, and Yee Whye Teh. 2019. Set transformer: A framework for attention-based permutation-invariant neural networks. In *International Conference on Machine Learning*. PMLR, 3744–3753.

[35] Georg Martius and Christoph H Lampert. 2016. Extrapolation and learning equations. *arXiv preprint arXiv:1610.02995* (2016).

[36] Trent McConaghy. 2011. FFX: Fast, scalable, deterministic symbolic regression technology. In *Genetic Programming Theory and Practice IX*. Springer, 235–260.

[37] James McDermott, David R. White, Sean Luke, Luca Manzoni, Mauro Castelli, Leonardo Vanneschi, Wojciech Jaskowski, Krzysztof Krawiec, Robin Harper, Kenneth De Jong, and Una-May O'Reilly. 2012. Genetic Programming Needs Better Benchmarks *(GECCO '12)*. Association for Computing Machinery, New York, NY, USA, 791–798.

[38] Robert I. McKay, Nguyen Xuan Hoai, Peter Alexander Whigham, Yin Shan, and Michael O'Neill. 2010. Grammar-based Genetic Programming: a survey. *Genetic Programming and Evolvable Machines* 11, 3 (Sept. 2010), 365–396.

[39] Julian Francis Miller. 2019. Cartesian genetic programming: its status and future. *Genetic Programming and Evolvable Machines* (Aug. 2019), 1–40.

[40] Julian Francis Miller and Simon L. Harding. 2008. Cartesian Genetic Programming. In *Proceedings of the 10th Annual Conference Companion on Genetic and Evolutionary Computation (GECCO '08)*. ACM, New York, NY, USA, 2701–2726.

[41] Alberto Moraglio, Krzysztof Krawiec, and Colin G. Johnson. 2012. Geometric Semantic Genetic Programming. In *Parallel Problem Solving from Nature - PPSN XII*. Vol. 7491. Springer Berlin Heidelberg, Berlin, Heidelberg, 21–31.

[42] T Nathan Mundhenk, Mikel Landajuela, Ruben Glatt, Claudio P Santiago, Daniel M Faissol, and Brenden K Petersen. 2021. Symbolic Regression via Neural-Guided Genetic Programming Population Seeding. In *Advances in Neural Information Processing Systems*.

[43] Luis Muñoz, Leonardo Trujillo, Sara Silva, Mauro Castelli, and Leonardo Vanneschi. 2019. Evolving multidimensional transformations for symbolic regression with M3GP. *Memetic Computing* 11, 2 (2019), 111–126.

[44] Randal S. Olson, William La Cava, Patryk Orzechowski, Ryan J. Urbanowicz, and Jason H. Moore. 2017. PMLB: a large benchmark suite for machine learning evaluation and comparison. *BioData Mining* 10, 36 (11 Dec 2017), 1–13.

[45] 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.

[46] Brenden K Petersen, Mikel Landajuela Larma, Terrell N Mundhenk, Claudio Prata Santiago, Soo Kyung Kim, and Joanne Taery Kim. 2021. Deep symbolic regression: Recovering mathematical expressions from data via risk-seeking policy gradients. In *International Conference on Learning Representations*.

[47] Subham Sahoo, Christoph Lampert, and Georg Martius. 2018. Learning equations for extrapolation and control. In *International Conference on Machine Learning*. PMLR, 4442–4450.

[48] Michael Schmidt and Hod Lipson. 2009. Distilling Free-Form Natural Laws from Experimental Data. *Science* 324, 5923 (2009), 81–85.

[49] Leonardo Trujillo, Luis Muñoz, Edgar Galván-López, and Sara Silva. 2016. neat genetic programming: Controlling bloat naturally. *Information Sciences* 333 (2016), 21–43.

[50] Silviu-Marian Udrescu, Andrew Tan, Jiahai Feng, Orisvaldo Neto, Tailin Wu, and Max Tegmark. 2020. AI Feynman 2.0: Pareto-optimal symbolic regression exploiting graph modularity. *arXiv preprint arXiv:2006.10782* (2020).

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

[52] Nguyen Quang Uy, Nguyen Xuan Hoai, Michael O'Neill, Robert I McKay, and Edgar Galván-López. 2011. Semantically-based crossover in genetic programming: application to real-valued symbolic regression. *Genetic Programming and Evolvable Machines* 12, 2 (2011), 91–119.

[53] Marco Virgolin, Tanja Alderliesten, and Peter AN Bosman. 2019. Linear scaling with and within semantic backpropagation-based genetic programming for symbolic regression. In *Proceedings of the genetic and evolutionary computation conference*. 1084–1092.

[54] Ekaterina J Vladislavleva, Guido F Smits, and Dick Den Hertog. 2008. Order of nonlinearity as a complexity measure for models generated by symbolic regression via pareto genetic programming. *IEEE Transactions on Evolutionary Computation* 13, 2 (2008), 333–349.

[55] Sanford Weisberg. 2005. *Applied linear regression*. Vol. 528. John Wiley & Sons.

[56] Tony Worm and Kenneth Chiu. 2013. Prioritized grammar enumeration: symbolic regression by dynamic programming. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*. 1021–1028.

[57] Hengrui Xing, Ansaf Salleb-Aouissi, and Nakul Verma. 2021. Automated Symbolic Law Discovery: A Computer Vision Approach. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 35. 660–668.

[58] Jinghui Zhong, Yusen Lin, Chengyu Lu, and Zhixing Huang. 2018. A deep learning assisted gene expression programming framework for symbolic regression problems. In *International Conference on Neural Information Processing*. Springer, 530–541.