Taylor Genetic Programming for Symbolic Regression

Baihe He
Beijing Key Laboratory of Petroleum Data Mining, China University of Petroleum
Beijing, China
hebaihe@hotmail.com

Qiang Lu
Beijing Key Laboratory of Petroleum Data Mining, China University of Petroleum
Beijing, China
luqiang@cup.edu.cn

Qingyun Yang
Beijing Key Laboratory of Petroleum Data Mining, China University of Petroleum
Beijing, China
yangqingyun.cup@hotmail.com

Jake Luo
Department of Health Informatics and Administration, University of Wisconsin Milwaukee
Milwaukee, United States
jakeluo@uwm.edu

Zhiguang Wang
Beijing Key Laboratory of Petroleum Data Mining, China University of Petroleum
Beijing, China
cwangsg@cup.edu.cn

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

GECCO ’22, July 9–13, 2022, Boston, MA, USA
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9237-2/22/07...$15.00
https://doi.org/10.1145/3512290.3528757

1 INTRODUCTION

Symbolic regression (SR) refers to finding a symbolic equation $f_\theta$ 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 $8.2 \times 10^{162}$ [25]. Therefore, finding a good symbolic equation for the data from the huge possible search space is a challenging task.

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, 26, 36, 44], 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 $f_\theta$ 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 $\theta$ 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, 22, 28, 33, 42, 47] find parameters $\theta$ 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 $f_\theta$s, the results are commonly biased by the pre-defined regression model $f$s 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.

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 [33], GSGP [38], BSR [18], SVM, and XGBoost, on the three types of benchmarks: classical SR [34], AIFeynman [47], and Penn machine learning benchmarks [41].

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 [50], SVM [10], XGBoost [8], and neural network (NN) [30], 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 ($\theta$) 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 $\theta$. Neural networks [2, 23, 28, 42, 43] 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 ($X, Y, F$). For example, GrammarVAE (GVAE) [28] trains a variational autoencoder [23] to directly encode from and decode to the parse trees using a context-free grammar. Recently, DSR [42] 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 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 distribution and uses BFGS to optimize these constants.

Besides the above neural networks for SR, EQL [32, 43] designs a shallow fully-connected neural network where standard activation functions (e.g., tanh, ReLU) are replaced with basic functions (e.g., $+, \times, \sin$). Once the neural network is trained, it can represent a symbolic equation fitted to the given dataset. AIFeynman [46, 47] employs neural networks to map the given dataset into simplifying
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 [19]. 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 [26] is still a commonly used approach to deal with SR. GP uses evolutionary operators – crossover, mutation, and selection, to change the individual encoding and generate better offspring for searching a solution in the mathematical expression space. Various GP’s employ different individual encodings to represent mathematical equations, such as tree-encoded GPs [7, 26, 35, 38, 45], graph-encoded GPs [37, 44], and linearly encoded GPs [3, 14, 31].

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 $8.2 \times 10^{162}$. In comparison, without real constants, its size is $1.054 \times 10^{16}$ [25]. So, some GP methods [24, 27, 51] 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 [24, 25] uses the abstract expression grammar to represent the skeleton of a mathematical expression and utilizes PSO [21] to find constant values. Unlike AEG-GP, PGE [51] 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 [27], MRGP [1], FEW [29], MSGP [40], 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 [33] 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 $a_i$ in $y = a_0 + \sum_{i=1}^{N} a_i \times B_i(x)$' by using pathwise regularized learning.

The other research line uses the hybrid of a neural network and a GP, called DL-GP. DL-GPs [11, 39, 52, 53] leverage a neural network to obtain features from the given dataset and apply these features to guide GP. For example, Xing et al. [52] 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-GP’s 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.

3 TAYLOR FEATURES ANALYSIS

3.1 Obtaining a Taylor polynomial

Taylor’s theorem [17] states that if a function $f$ has $n+1$ continuous derivatives on an open interval containing $a$, for each $x$ in the interval,

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

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.

\[
(\frac{x_1 - x_0}{k}) f' ((x_0)) + \ldots + (\frac{x_k - x_0}{k}) f' (x_k) \approx f' (x_1) - f (x_0)
\]

(\frac{x_2 - x_0}{k}) f' ((x_0)) + \ldots + (\frac{x_k - x_0}{k}) f' (x_k) \approx f' (x_2) - f (x_0)

\ldots

(\frac{x_k - x_0}{k}) f' (x_k) \approx f' (x_k) - f (x_0)

(2)

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

\[
F \approx DA^{-1}
\]

, 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 $a_i \in A$, $a_{ij} = \frac{(x_i - x_0)^j}{j!}$. $F$ 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 [16], 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 $C_{n+k}^n - 1 < 1$
[\|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 \((C^n_k - 1) \times (C^n_k - 1)\) 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, monotonity 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 "*" that lets \( f(x_1, ..., x_n) = f_1(x_1, ..., x_k) \) \( \circ f_2(x_m, ..., x_p) \) where the two variable sets, \( \{x_1, ..., x_k\} \) and \( \{x_m, ..., x_p\} \), both belong to \( \{x_1, ..., x_n\} \), \( \{x_1, ..., x_k\} \cap \{x_m, ..., x_p\} = \phi \), it is called "*" separability. The separability property can decompose a complex multivariate function into multiple simple functions.

If "*" 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 \( 1.1x + 0.2x^2 + 1.5sin(y) - 3.7 \). While, if \( f \) is multiplication separability, then the function that the Taylor polynomial approximates is multiplication separability. Because, if a function is multiplication separability, i.e., \( f(x_1, ..., x_n) = f_1(x_1, ..., x_k) \times f_2(x_m, ..., x_p) \), then \( log f(x_1, ..., x_n) = log f_1(x_1, ..., x_k) + log f_2(x_m, ..., x_p) \). 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 \([x_0, x_h]\). 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 \([-1.1, 1.1] \). The boundary of the Taylor polynomial \( 1.1x + 0.2x^2 - 3.7 \) is \([-1.1, 1.1] \). While, if \( x_0 \) and \( x_h \) the function that the dataset represents is a monotonically increasing function. Otherwise, if \( y_i \geq y_j \) and \( x_i \leq x_j \), the function is a monotonically decreasing function.

3.2.4 Monotonic. For all points \((x_i, y_i)\) in a dataset \((X, Y)\), if \( y_i \geq y_j \) and \( x_i \leq x_j \), the function that the dataset represents is a monotonically increasing function. Otherwise, if \( y_i \geq y_j \) and \( x_i \leq x_j \), the function is a monotonically 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, monotonity, and parity. FFEM mainly contains two evolvable operators, individual initialization (initIndividualByFeatures) and individual recombination (recombinByFeatures). 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 \( \alpha \) utilizes the individual initialization to produce offspring with the probability \( \beta \), 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,
Algorithm 1 function feature evolution method

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

Output: best
1: P ← initIndividualByFeatures(F,popSize=N)
2: best ← selectBestIndividual(P)
3: while best.fitness ≤ threshold and g < maxGen do
   4:      for all i = 1 to N do
   5:         p_1, p_2 ← randomSelectTwoIndividuals(P)
   6:         if rand() < α then
   7:            child ← recombineByFeatures(p_1, p_2,F)
   8:         else if rand() < α + β then
   9:            child ← initIndividualByFeatures(F,popSize=1)
  10:       else
  11:          child ← p_1
  12:       end if
  13:       nextP[i] ← child
  14:   end for
  15:   P ← nextP
  16:   best ← min(best, selectBestIndividual(P,(X_i, Y_i)))
  17:   g ++
  18: end while
  19: return best

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 also shows a sub-space that contains all mathematical expressions expanded from the tree. Moreover, all trees with depth 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 "c + sin x", "c + sin x + x", and "sin x + c x". The sub-space "c + sin x" is a mathematical expression "c + x + sin(x)", which contains all mathematical expressions expanded from "c + sin x", such as "c + sin x + 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, +, −, ×, ÷, a^n, ln, ...), the boundary of the sub-space is [−∞, ∞]. Otherwise, it is computed by interval arithmetic. For example, for the sub-space "c + sin x", there is a path "x + c", so its boundary is [−∞, ∞]. There is no such path for the sub-space "c + sin x", 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 ≥ 0 (or d ≤ 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 "c + sin x + 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 monotonicity 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], odd}, for the selected sub-space "c + sin x + 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 "c + sin x + x + 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 {+, −, ×, ÷, f(g(x))}. With these steps the method finally generates an individual "2x + 2sin(x)" whose boundary contains [−10, 10].

| op   | g(x) | results |
|------|------|---------|
| +    | odd  | odd     |
| +    | even | even    |
| −    | odd  | odd     |
| −    | even | even    |
| ×    | /    | odd     |
| /    | even | even    |
| /    | odd  | odd     |
| f(g(x)) | even | even |
| f(g(x)) | odd  | even   |
| f(g(x)) | even | odd    |
| f(g(x)) | odd  | 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 recomposed 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 recomposed 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) [34], Penn
Machine Learning Benchmarks (PMLB) [41], and Feynman Symbolic Regression Benchmarks (FSRB) [47]. SRB consists of twenty-three SR problems derived from the five canonical symbolic regression benchmarks, Nguyen [48], Korns [24], Koza [9], Keijzer [20], and Vladislavleva [49]. PMLB includes seven regression tasks and three classification tasks. FSRB contains forty-eight Feynman equations in [47]. 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.

![Properties of Benchmarks](image)

**Figure 2: Properties of Benchmarks.**

### 5.2 Algorithm Parameter Settings

We compare TaylorGP with two kinds of baseline algorithms: four symbolic regression methods and five machine learning methods. The symbolic regression methods include GPlearn, FFX, geometric semantic genetic programming (GSGP)[38] and bayesian symbolic regression (BSR) [18]. 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.

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

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.

![Normalized $R^2$ scores of the ten algorithms](image)

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

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.

### 6.2 Discussion

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

---

**Table 2: Algorithm parameters**

| Name          | Parameter                | Value          |
|---------------|-------------------------|----------------|
| TaylorGP      | Function Set            | $+,\times,\sin,\cos,\log,\exp,\sqrt{\cdot}$ |
|               | 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            | $+,\times,\sin,\cos,\log,\exp,\sqrt{\cdot}$ |
|               | Max generations         | 10000          |
|               | Population Size         | 1000           |
|               | Crossover Rate          | 0.7            |
|               | Mutation Rate           | 0.2            |
|               | Stopping Threshold      | 1e-5           |
| BSR           | Function Set            | $+,\times,\sin,\cos,\log,\exp,\sqrt{\cdot}$ |
|               | Max Generations         | 10000          |
|               | Population Size         | 1000           |
|               | Crossover Rate          | 2              |
|               | Stopping Threshold      | 1e-5           |
| FFX           | None                    |                |
| LR            | Normalize               | FALSE          |
| KR            | Kernal                  | linear, ‘polynomial’, ‘rbf’, ‘sigmoid’ |
|               | Gamma                   | 0.01, 0.1, 1.1 |
|               | Regularization          | 0.001, 0.1     |
| RF            | Number of Estimators    | 10, 100, 1000  |
|               | Max Features            | sqrt, log, None |
| SVM           | Kernal                  | linear, ‘polynomial’, ‘rbf’, ‘sigmoid’, ‘precomputed’ |
|               | Gamma                   | 0.0001, 0.01, 0.05, 0.1, 0.2 |

---
pared with GPLearn, GSGP and BSR, TaylorGP can find the optimal result at the point \((2,2)\). According to the Taylor polynomial, the function
\[
e^{x_0^2} \times \sin(x_0^3) + 2^x_0 (\exp(\log(x_0) x_1)) \exp(\sin(x_0^3) + (\exp(\log(x_0) x_1)))
\]

they run for 10,000 generations. TaylorGP first generates the Taylor polynomial
\[
U_2 = \frac{1}{2} k_{spring} x^2
\]
used in the following evaluation. One benchmark is the \(\exp(x_0)\), and 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 \(x_0^{X1}\).](image)

For the benchmark \(x_0^{X1}\), where \(x_0\) and \(x_1\) are in \([2, 4]\), compared with GPLearn, GSGP and BSR, TaylorGP can find the optimal result \(\exp(\log(x_0) x_1) = x_0^{X1}\) at the 1000th generation. While the other three algorithms still cannot find the optimal results until they run for 10,000 generations. TaylorGP first generates the Taylor polynomial
\[
U_2 = \frac{1}{2} k_{spring} x^2
\]
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.](image)

For the Feynman benchmark \(U = \frac{1}{2} k_{spring} x^2\), TaylorGP finds the optimal results at the 9th generation, because TaylorGP can directly obtain \(0.2 k_{spring} x^2\) owing to the low polynomial discrimination. TaylorGP achieves the Taylor polynomial \(U_2 = (1.454e - 6)x_0^3 - (1.181e - 6)x_0^2 x_1 - (7.784e - 6)x_0 x_1^2 + (0.5)x_0 x_1^3 + (5.809e - 6)x_0 x_1 + (9.249e - 6)x_0 - (2.53e - 6)x_1^2 + (2.041e - 5)x_1^3 - (6.481e - 5)x_1 + (5.579e - 5)\) from the benchmark. After omitting insignificant coefficients that are less than \(e - 4\), the Taylor polynomial is \(0.5x_0 x_1^2\). As the RMSE of \(0.5x_0 x_1^2\) is less than the stopping threshold \(e - 5\), it is the final result that TaylorGP finds.

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 R² 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 datasets. 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 cannot contain some terms whose coefficients are less than the threshold. 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 \( \frac{1}{6} x - \frac{1}{120} x^3 + \frac{1}{5040} x^5 - \frac{1}{30240} x^7 + \ldots \). However, according to Equation 3, TaylorGP sets a 4-order Taylor polynomial and obtains the polynomial \( 0.015 + \frac{1}{7} x + 0.003 x^2 - \frac{1}{3} x^3 + \ldots \) 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, ‘\( x^2 \)’ and ‘\( U = \frac{1}{2} k \text{spring}_x^2 \)’. So, TaylorGP can utilize the Taylor features to reduce its search space and speed up its search.

7 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.

ACKNOWLEDGMENTS

This work is supported by China National Key Research Project (No.2019YFC0312003)
