DIGEST: DISCOVERY-GUIDED LLM TUNING FOR SYMBOLIC REGRESSION

Anonymous authors
Paper under double-blind review

ABSTRACT

Symbolic regression (SR) seeks to discover mathematical expressions directly from data, without requiring task-specific scientific descriptions. Despite tremendous progress in both conventional search-based methods and recent LLM-based methods, two critical limitations largely remain. First (*Limited Generation Flexibility*), conventional search-based methods rely on either fixed search rules or specialized learned generators, severely limiting adaptive candidate generation across diverse expression structures. Second (*Underutilized Search Discoveries*), most LLM-based methods cannot reliably recover ground-truth expressions because they fall short in effectively leveraging the LLM itself to thoroughly digest past expressions or adaptively guide future search. To address these limitations, we propose DIGEST. To render greater flexibility in generation, DIGEST repurposes a pretrained decoder block from a lightweight LLM into a symbolic policy for Genetic Programming (GP) population seeding, thereby leveraging the broader modeling capacity of the LLM. To more thoroughly utilize search discoveries, high-quality expressions discovered by GP are fed back to update the LLM-derived policy online, thereby enabling the LLM to progressively internalize past search discoveries and translate them into improved subsequent population generation. Experiments show that on two ground-truth benchmarks, DIGEST outperforms the strongest competitor by up to 2.59% in recovery. Among LLM-based baselines, it improves recovery by at least 54.91% while using at most 17.4% of the strongest competitor's mean search time. On 122 BlackBox datasets, DIGEST achieves the highest accuracy while maintaining a moderate model complexity.

1 INTRODUCTION

Symbolic regression (SR) seeks to discover mathematical expressions directly from data, without requiring task-specific scientific descriptions (La Cava et al., 2021). By searching over both expression structure and numerical parameters, SR provides a way to construct explicit models when a suitable functional form is not known in advance. Recent large language model (LLM) approaches further use scientific descriptions and domain knowledge as auxiliary information for equation discovery (Shojaee et al., 2025a). Such information can be valuable when available. However, there are also black-box discovery settings in which only numerical input-output observations are available, without reliable task-specific descriptions or variable semantics. In such settings, SR is particularly valuable because candidate functional forms cannot be readily specified from domain knowledge and must instead be discovered from the observations themselves. Accordingly, SR methods, particularly LLM-based ones, should remain effective even when scientific context is unavailable, rather than relying on it as a prerequisite. We focus on this clas-

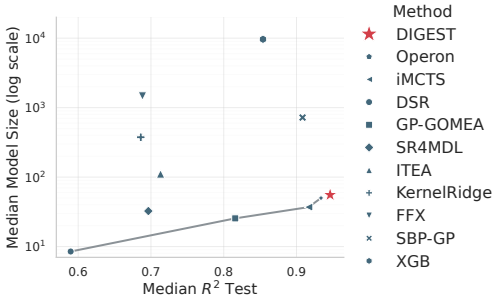

Figure 1: DIGEST achieves the highest R^2 with moderate model size on BlackBox datasets.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

sical data-driven SR setting, using input-output observations and a predefined symbolic vocabulary without supplied variable semantics or physical units.

Despite tremendous progress in both conventional search-based and recent LLM-based SR, two critical limitations remain. First (*Limited Generation Flexibility*), conventional search-based methods generate candidates through predefined search mechanisms or specialized learned generators. For example, PySR evolves expression populations through genetic operations (Cranmer, 2023), DSR learns a recurrent generation policy (Petersen et al., 2019), and E2E trains a Transformer on synthetic equation-data pairs (Kamienny et al., 2022). These mechanisms can exploit numerical feedback, but their candidate-generation behavior is shaped by the chosen search rules or SR-specific learned priors. This can limit generation flexibility across targets with different operator combinations, nested functions, and variable interactions.

Second (*Underutilized Search Discoveries*), most LLM-based methods fail to reliably recover ground-truth expressions because they fall short in effectively leveraging the LLM itself to thoroughly digest past expressions or adaptively guide future search. Although many LLM-based methods reuse previously discovered expressions and structures together with their evaluation scores, these search discoveries are typically incorporated through prompts, memories, libraries, or search operators while the underlying LLM remains fixed (Merler et al., 2024; Shojaee et al., 2025a; Grayeli et al., 2024). As a result, past discoveries can influence subsequent search, but are not progressively internalized by the LLM itself to reshape future candidate generation.

To address these limitations, we propose DIGEST (Discovery-Guided LLM Tuning for Symbolic Regression via Genetic Programming), which turns discoveries from genetic programming (GP) into online training signals for an LLM-derived symbolic policy. To render greater flexibility in generation, DIGEST repurposes a pretrained decoder block from a lightweight LLM into a symbolic policy for GP population seeding. This LLM-derived policy maps expression-tree context to distributions over symbolic tokens. By leveraging the broader modeling capacity of the LLM, the policy supports more flexible candidate generation across diverse expression structures. The generated expressions initialize GP populations for multi-generation evolutionary refinement. To more thoroughly utilize search discoveries, high-quality expressions discovered by GP are fed back to adapt the LLM policy online. The adapted policy then seeds the next GP search. Through this online adaptation, the policy progressively internalizes past search discoveries and translates them into improved subsequent population generation.

Experiments on Nguyen and Livermore show that DIGEST outperforms the strongest competitor by up to 2.59% in recovery. Among the evaluated LLM-based baselines, it improves recovery by at least 54.91% while using at most 17.4% of the mean search time of the strongest competitor. On 122 SRBench BlackBox datasets, DIGEST achieves the highest cross-dataset median test R^2 among the compared methods while maintaining moderate model complexity. Our main contributions are summarized as follows:

- **LLM-derived symbolic policy.** We repurpose a pretrained decoder block from a lightweight LLM into a symbolic policy that maps expression-tree context to distributions over symbolic tokens for GP population seeding. By leveraging the broader modeling capacity of the LLM, this design supports more flexible candidate generation across diverse expression structures.
- **Online adaptation from search discoveries.** We feed high-quality expressions discovered by GP back to update the LLM policy online. This allows the LLM to progressively internalize past search discoveries and use them to improve subsequent population generation.
- **Evaluation.** Experiments on two ground-truth benchmarks and 122 SRBench BlackBox datasets demonstrate strong recovery and predictive performance (Figure 1).

2 PRELIMINARIES

Notation. We use calligraphic capitals for sets and collections, bold lowercase letters for vectors, and bold uppercase letters for matrices. Table 1 summarizes the main notation.

Table 1: Main notation.

Symbol	Meaning
$\mathcal{D}, \mathcal{A}$	Observations and symbolic vocabulary.
ϕ, ϕ_t, L	Expression sequence, its token at step t, and length.
p_t, s_t	Parent and sibling tokens before generation step t.
π_θ, F_θ	Symbolic policy and pretrained decoder block.
$\mathbf{E}_p, \mathbf{E}_s$	Parent and sibling embedding tables.
$\mathbf{W}_o, \mathbf{b}_o$	Symbolic output projection and bias.
$R(\phi), \mathbf{m}_t$	Training reward and generation-constraint vector.
N, S, M	Population size, GP generations per restart, and elite limit.
$\mathcal{B}, \mathcal{E}, \mathcal{C}$	Generated batch, GP elites, and combined feedback batch.
$\mathcal{C}_\epsilon, \mathcal{Q}$	High-reward feedback batch and persistent priority queue.
K, ϵ	Queue capacity and reward-tail parameter.
B, b	Nominal evaluation budget and accumulated evaluation count.

Symbolic regression and expression trees. Given training observations $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$, with $\mathbf{x}_i \in \mathbb{R}^d$, symbolic regression (SR) searches for an expression whose predictions fit the observed relationship. Let $\mathcal{A}$ be a predefined vocabulary of operators, input variables, and optional free-constant tokens. Free constants are coefficients fitted to the observations. No task-specific scientific descriptions, variable semantics, or physical units are supplied. An expression tree has operators at internal nodes and variables or constants at leaves. An operator's *arity* is its number of inputs: addition has two inputs and sine has one. We represent an expression by its pre-order sequence $\phi = (\phi_1, \ldots, \phi_L)$, visiting each node before its children, from left to right. Each $\phi_t \in \mathcal{A}$ labels one node, and $L = |\phi|$ is the number of nodes. A complete sequence and the known arities determine the tree. For example, $\sin(x_1) + x_2$ corresponds to $\phi = (+, \sin, x_1, x_2)$. We write f_ϕ for the resulting function, after fitting free constants when present. Its training-data reward $R(\phi)$ measures fit, with larger values indicating better predictions.

Genetic programming. Genetic programming (GP) searches over expression populations using fitness-based selection, subtree crossover, and mutation (Koza, 1994). These operations select promising candidates, recombine subexpressions, and introduce structural variations, respectively.

3 METHOD

We present DIGEST through three stages: *expression generation* with an LLM-derived symbolic policy (Section 3.1), *candidate refinement* through GP (Section 3.2), and *adaptation from search discoveries* to improve subsequent population generation (Section 3.3). These stages repeat across GP restarts. Figure 2 illustrates the framework, and Algorithm 1 summarizes the procedure.

3.1 EXPRESSION GENERATION

To render greater flexibility in generation, DIGEST repurposes a pretrained decoder block from a lightweight LLM into a symbolic policy π_θ for GP population seeding, where θ denotes its trainable parameters. Compared with conventional search mechanisms or SR-specific generators, the policy leverages the broader modeling capacity of LLM to model diverse expression structures.

Expression-tree context. Symbolic expressions are naturally represented as trees, whereas autoregressive LLMs operate on sequences. We therefore encode the evolving expression tree as a sequence of local structural contexts along its pre-order traversal. This converts tree construction into a sequential prediction problem that can be naturally modeled by an autoregressive LLM. Before generating the next symbolic token ϕ_t, the generated prefix $\phi_{<t}$ determines its parent p_t, the operator to which the next node attaches. The generated prefix $\phi_{<t}$ also determines its sibling s_t, which is the root of the previously completed sibling subtree. We use EMPTY when the parent or sibling context is absent. For the running example $\sin(x_1) + x_2$, Table 2 shows the structural context at each generation step, and Figure 3 shows the corresponding expression tree.

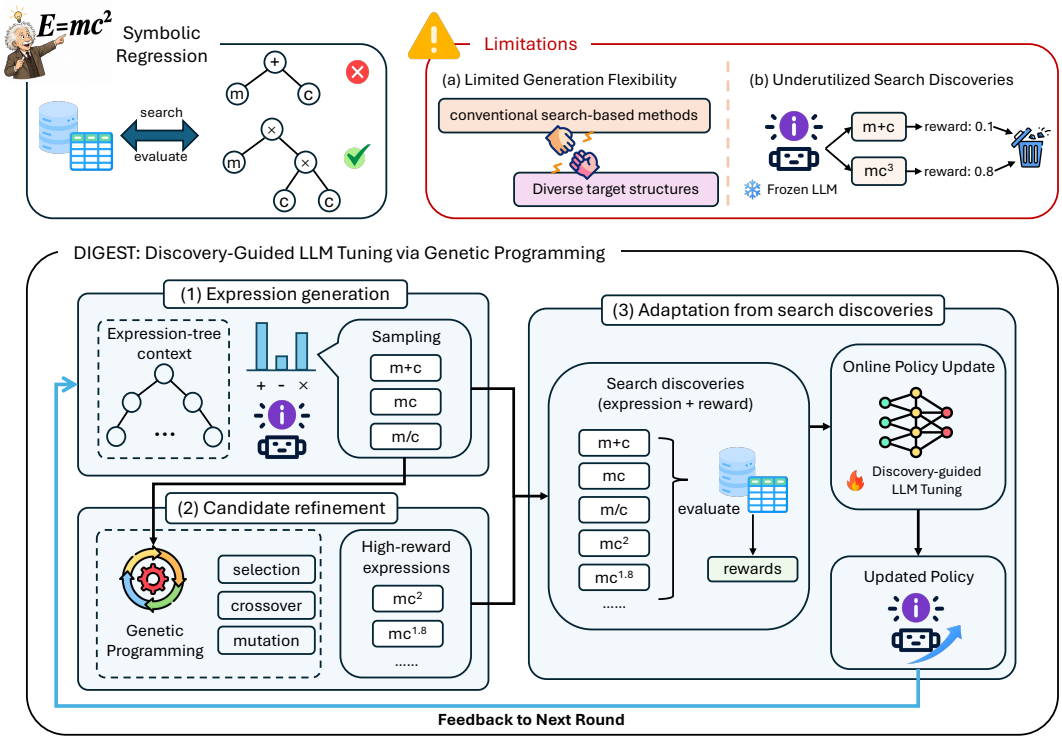

Figure 2: Overview of DIGEST. **Expression generation:** an LLM-derived symbolic policy samples a population from expression-tree context. **Candidate refinement:** GP searches for higher-reward expressions through evolution. **Adaptation from search discoveries:** policy-generated expressions and GP elites form a feedback batch, from which high-reward expressions provide training signals for online policy adaptation. The adapted policy seeds the next GP restart.

Table 2: Parent and sibling context for $\sin(x_1) + x_2$.

Generation step t	1	2	3	4
Next token ϕ_t	$+$	$\sin$	x_1	x_2
Parent p_t	EMPTY	$+$	$\sin$	$+$
Sibling s_t	EMPTY	EMPTY	EMPTY	$\sin$

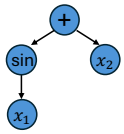

Figure 3: Expression tree for $\sin(x_1) + x_2$.

At the first step, both the parent and sibling contexts are EMPTY. Before generating token x_2, the parent is $+$ and the sibling is $\sin$, which is the root of the completed subtree $\sin(x_1)$ (Figure 3). Thus, the structural context can be constructed incrementally from the generated prefix.

Pretrained symbolic interface. The structural contexts above are defined over the symbolic vocabulary $\mathcal{A}$, whereas the pretrained decoder is learned over text tokens. We bridge the two spaces by initializing a symbolic input/output interface from pretrained token representations. For each symbol $a \in \mathcal{A}$, we assign a fixed textual label $c(a)$, such as $c(\texttt{mul}) = \texttt{multiply}$. If the tokenizer maps this label to token IDs $z_{a,1}, \ldots, z_{a,m_a}$, its symbolic representation is initialized as $\mathbf{e}_a = \frac{1}{m_a} \sum_{j=1}^{m_a} \mathbf{E}_{\mathrm{LM}}[z_{a,j}]$, where $\mathbf{E}_{\mathrm{LM}}$ is the pretrained token-embedding table. These representations initialize the parent and sibling embeddings $\mathbf{E}_p, \mathbf{E}_s$ and the symbolic output projection $\mathbf{W}_o$. The resulting components are separately parameterized and jointly optimized with the pretrained decoder block. Appendix A.1 provides the complete initialization details.

From structural context to symbolic prediction. The parent-sibling contexts are interleaved into a structural sequence

$$\mathbf{U}_t = [\mathbf{E}_p[p_1], \mathbf{E}_s[s_1], \ldots, \mathbf{E}_p[p_t], \mathbf{E}_s[s_t]]. \tag{1}$$

The pretrained decoder block F_θ transforms this sequence. Because each generation step contributes a parent and a sibling position, we use the hidden state at the current sibling position, $\mathbf{h}_{2t} = [F_\theta(\mathbf{U}_t)]_{2t}$, to predict the next symbol. For the running example, after generating the prefix $(+, \sin, x_1)$, the context $(p_4, s_4) = (+, \sin)$ is appended before predicting token x_2.

The resulting symbolic policy is

$$\pi_\theta(\phi_t = a \mid \phi_{<t}) = \left[\mathrm{softmax}(\mathbf{W}_o \,\mathrm{LN}(\mathbf{h}_{2t}) + \mathbf{b}_o + \mathbf{m}_t)\right]_a , \tag{2}$$

where LN is the trainable output LayerNorm, $\mathbf{b}_o$ is the symbolic output bias, and $\mathbf{m}_t$ is the additive mask for the generation constraints. Repeating this autoregressive prediction constructs a complete expression. At each restart, DIGEST samples N expressions $\mathcal{B} = (\phi^{(1)}, \ldots, \phi^{(N)})$, which form the starting population for GP refinement.

3.2 Candidate Refinement

Generated expressions provide starting hypotheses, but language pretraining alone does not optimize them for the observed data. We therefore place GP between generation and adaptation as a numerically guided refinement stage. Inspired by Mundhenk et al. (2021), GP starts from the policy-generated population and explores structural variations under task-specific numerical fitness. Rather than asking the LLM to supply every structural improvement through generation, this stage uses multi-generation evolutionary search to turn numerical feedback into high-quality search discoveries. These discoveries provide training signals for the policy adaptation described in Section 3.3.

GP refinement. At each restart, the generated batch $\mathcal{B}$ initializes a fresh GP population. RefineGP in Algorithm 1 evolves it through S generations of tournament selection, subtree crossover, and mutation. During this search, numerically valid candidates receive the training reward $R(\phi) = 1/(1 + \mathrm{NRMSE}(\phi))$, where NRMSE is the normalized root-mean-square error on the training observations (Appendix A.3). A hall-of-fame archive records high-reward expressions encountered during the search. The routine returns up to M archived expressions as an elite batch $\mathcal{E}$, together with the GP evaluation count n_{GP}.

Search discoveries. After refinement, we form the feedback batch $\mathcal{C} = \mathrm{Concat}(\mathcal{B}, \mathcal{E})$ and obtain its training rewards. Thus, $\mathcal{C}$ contains both expressions proposed by the policy and high-quality expressions discovered through GP refinement, providing the search discoveries used for online adaptation in the next stage.

3.3 Adaptation from Search Discoveries

To more thoroughly utilize search discoveries, we turn high-reward expressions into training targets for the LLM-derived policy. Rather than reusing past discoveries only as inference-time context for a fixed LLM, as in most existing LLM-based SR methods such as Merler et al. (2024); Shojaee et al. (2025a), we incorporate them through parameter updates that shape subsequent generation.

Retaining search discoveries. A persistent priority queue $\mathcal{Q}$ keeps up to K of the highest-reward distinct expressions submitted across restarts. For the current feedback batch $\mathcal{C}$, we form the high-reward batch $\mathcal{C}_\epsilon = [\phi \in \mathcal{C} : R(\phi) \geq q_{1-\epsilon}]$, where $q_{1-\epsilon}$ is the empirical $(1-\epsilon)$ quantile of its rewards. Only the highest-reward member of $\mathcal{C}_\epsilon$ is submitted to $\mathcal{Q}$.

Online policy adaptation. We adapt the LLM-derived policy toward retained high-reward expressions using priority queue training (PQT) (Abolafia et al., 2018), with auxiliary entropy regularization to encourage exploration. For one expression $\widetilde{\phi}$ sampled uniformly from $\mathcal{Q}$, we minimize

$$\mathcal{L}(\theta) = -w_{\mathrm{PQT}} \log \pi_\theta(\widetilde{\phi}) - \frac{\lambda_H}{|\mathcal{C}_\epsilon|} \sum_{\phi \in \mathcal{C}_\epsilon} H_\theta(\phi). \tag{3}$$

Here the expression log-likelihood is $\log \pi_\theta(\phi) = \sum_{t=1}^{|\phi|} \log \pi_\theta(\phi_t \mid \phi_{<t})$, and $H_\theta(\phi)$ sums conditional token entropies along the expression's valid positions. The weights w_{PQT} and λ_H balance likelihood training on the queue sample and entropy regularization over the current $\mathcal{C}_\epsilon$. Queue and loss details are provided in Appendix A.2.

At each restart, one AdamW step jointly updates the pretrained decoder block and symbolic interface (Appendix A.4). The adapted policy seeds the next GP run, while the persistent queue retains earlier discoveries for subsequent updates.

Algorithm 1 DIGEST: expression generation, candidate refinement, and adaptation from search discoveries

Require: Data $\mathcal{D}$; vocabulary $\mathcal{A}$; generation constraints, GP settings, and stopping criterion
Require: Pretrained token embeddings and a decoder block; N, S, M, K, ϵ, and nominal budget B
Ensure: Best feedback expression $\phi^\star$

Initialize π_θ, the symbolic interface, and AdamW
$\mathcal{Q} \leftarrow \varnothing$; $\phi^\star \leftarrow$ None; $b \leftarrow 0$; clear the expression cache
for $r = 1, \ldots, \lfloor B/N \rfloor$ **do**
Expression generation
  Sample $\mathcal{B} = (\phi^{(1)}, \ldots, \phi^{(N)})$ from π_θ
Candidate refinement
  $(\mathcal{E}, n_{\mathrm{GP}}) \leftarrow \textsc{RefineGP}(\mathcal{B}, \mathcal{D}, S, M)$
  $\mathcal{C} \leftarrow \mathrm{Concat}(\mathcal{B}, \mathcal{E})$
  Obtain training rewards for $\mathcal{C}$, reusing cached evaluations
  Update $\phi^\star$ using rewards in $\mathcal{C}$; $b \leftarrow b + N + n_{\mathrm{GP}}$
Adaptation from search discoveries
  $\mathcal{C}_\epsilon \leftarrow [\phi \in \mathcal{C} : R(\phi) \geq q_{1-\epsilon}]$
  $\widehat{\phi} \leftarrow \arg\max_{\phi \in \mathcal{C}_\epsilon} R(\phi)$
  Submit $\widehat{\phi}$ to $\mathcal{Q}$; keep up to K highest-reward distinct submitted expressions
  Sample one $\widetilde{\phi}$ uniformly from $\mathcal{Q}$
  Apply one AdamW update using Eq. equation 3
  if $b > B$ or the configured stopping condition holds **then**
    break
  end if
end for
return $\phi^\star$

4 Experiments

4.1 Experimental Setup

Datasets and metrics. We evaluate DIGEST on the Nguyen (Petersen et al., 2019) and Livermore (Mundhenk et al., 2021) ground-truth datasets, containing 12 and 22 target expressions, respectively, and on 122 SRBench BlackBox datasets (La Cava et al., 2021). For ground-truth datasets, we report recovery averaged across equations, together with median test NMSE and mean search time for the LLM-based comparison. For SRBench BlackBox, we follow the standard SRBench evaluation protocol (La Cava et al., 2021) and report cross-dataset median test R^2 and model size. Detailed benchmark definitions and evaluation protocols are provided in Appendix B.

Baselines. For Nguyen and Livermore, we compare with two groups: (1) conventional SR methods, including DSR (Petersen et al., 2019), GEGL (Ahn et al., 2020), NGGP (Mundhenk et al., 2021), PySR (Cranmer, 2023), and iMCTS (Huang et al., 2026); and (2) LLM-based methods, including ICSR (Merler et al., 2024), LLM-SR (Shojaee et al., 2025a), LaSR (Grayeli et al., 2024), and RESTART (Li & Pan, 2026). For SRBench BlackBox, we compare with the 21 baseline methods reported by La Cava et al. (2021), together with iMCTS (Huang et al., 2026) and SR4MDL (Yu et al., 2025). TPSR (Shojaee et al., 2023) is evaluated separately under its BlackBox setting. Descriptions and implementation details of the baseline methods are provided in Appendix C.

Implementation. We instantiate DIGEST with a decoder block from Qwen2.5-0.5B (Yang et al., 2024), whereas the LLM-based baselines use 8B backbones. For Nguyen and Livermore, our evaluation uses 25 independent random seeds per equation. DIGEST and the conventional baselines are evaluated under a two-million-evaluation budget, while the LLM-based baselines use bounded, method-specific search budgets to keep search costs in a comparable range to DIGEST. All experiments run on one NVIDIA A40 GPU on the NCSA Delta cluster. Full configurations of DIGEST are provided in Appendix A.4; baseline settings and timing details are provided in Appendix C.

4.2 Results on Ground-Truth Benchmarks

State-of-the-art recovery. Table 3 shows that DIGEST achieves the highest average recovery on both Nguyen and Livermore. It reaches 93.33% on Nguyen and 74.00% on Livermore, surpassing the strongest conventional baseline, iMCTS, by 0.08 and 2.59 percentage points, and the strongest LLM-based baseline, LaSR, by 63.66 and 54.91 percentage points, respectively. These gains support the central design of DIGEST: turning discoveries from GP into online training signals for an LLM-derived symbolic policy. Per-equation results are provided in Appendix D.1.

Table 3: Average recovery rates (%) on Nguyen and Livermore. The best result in each row is bold.

	Conventional methods					LLM-based baselines				
Benchmark	DSR	GEGL	NGGP	PySR	iMCTS	ICSR	LLM-SR	LaSR	RESTART	DIGEST
Nguyen	83.58	86.00	92.33	74.41	93.25	14.00	3.33	29.67	4.00	**93.33**
Livermore	30.41	56.36	71.09	46.14	71.41	16.73	1.27	19.09	2.18	**74.00**

Further comparison with LLM-based methods. Figure 4 further compares DIGEST with the LLM-based baselines in prediction error and search time. DIGEST achieves the lowest median test NMSE on both benchmarks, with 1.50×10^{-32} on Nguyen and 5.20×10^{-32} on Livermore, and the shortest mean search times of 49.2 and 118.8 seconds, respectively. These times are only 7.2% and 17.4% of LaSR's mean search time. Notably, DIGEST repurposes a single pretrained decoder block from Qwen2.5-0.5B, whereas all four LLM-based baselines use 8B backbones (see Appendix C).

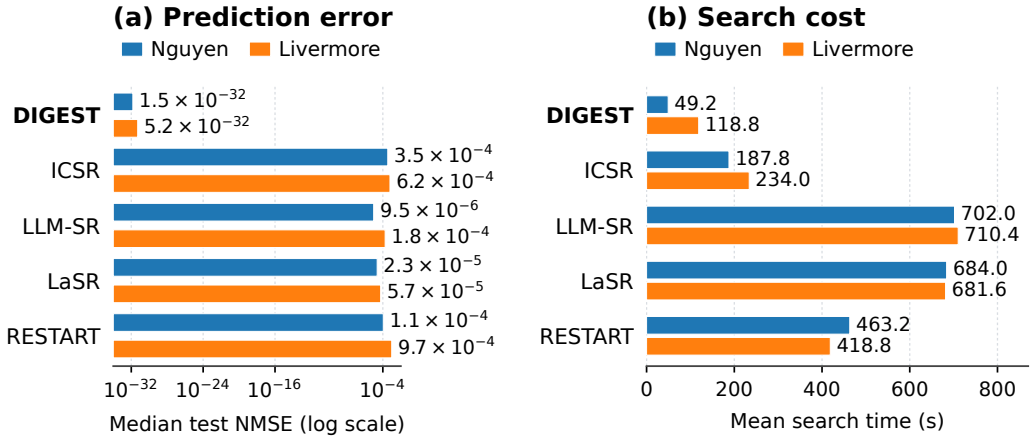

Figure 4: Further comparison of DIGEST with LLM-based baselines over 25 independent runs per equation on Nguyen and Livermore. Left: median test NMSE on a logarithmic scale. Right: mean search time in seconds.

4.3 Results on SRBench BlackBox

On the 122 SRBench BlackBox datasets, DIGEST achieves the highest cross-dataset median test R^2 among the compared methods, including the 21 SRBench baselines, iMCTS, and SR4MDL, while maintaining moderate model size (Figure 5). These results show that the benefit of LLM's learning from search discoveries extends beyond ground-truth equation recovery to black-box regression, where the target expression is unavailable. A Pareto view of the accuracy-complexity trade-off is provided in Appendix D.2 (Figure 6), where DIGEST lies on the Pareto frontier with the highest median test R^2. We also compare with TPSR on the 57-dataset subset of SRBench BlackBox used in its evaluation. DIGEST achieves a median test R^2 of 0.956 compared with 0.945 for TPSR, with mean model size 53.44 compared with 95.71. The SRBench evaluation protocol and the separate TPSR setting are detailed in Appendices B.2 and B.3.

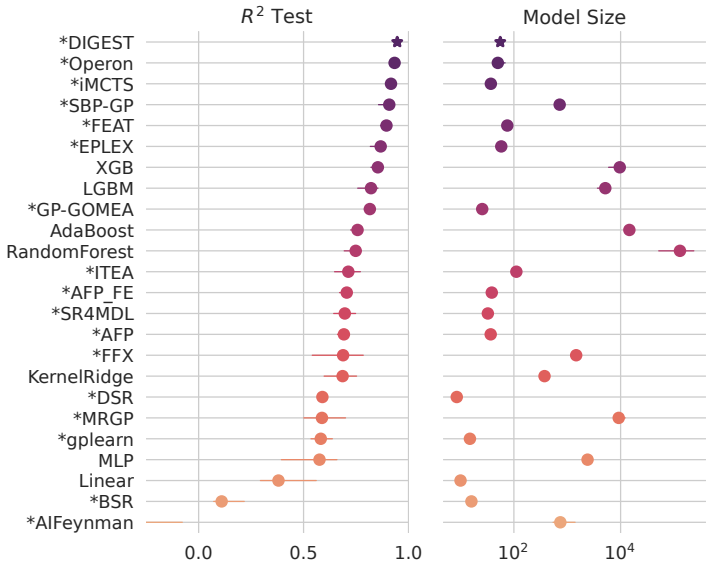

Figure 5: Performance on 122 SRBench BlackBox datasets. Left: cross-dataset median test R^2. Right: cross-dataset median model size. Asterisks identify symbolic regression methods. DIGEST achieves the highest median predictive accuracy while maintaining moderate model complexity.

5 Ablation Study

We study three effects over 25 independent runs per equation (Table 4), with all other evaluation settings following Section 4.1. For **online adaptation**, we compare frozen, interface-only, and decoder-updated policies. For **decoder depth**, we compare DIGEST's single-block policy with full-depth policies that execute all decoder blocks. For **population seeding**, we replace the LLM-derived policy with uniform random seeding while retaining the same GP refinement. Except for the targeted change in each ablation, all other components and settings of DIGEST are kept unchanged. For the full-depth controls, LoRA keeps the pretrained decoder weights frozen and learns low-rank updates across all decoder blocks, whereas full-dense tuning updates all decoder weights. Detailed configurations are provided in Appendices A.4 and A.5.

Table 4: Ablations of online adaptation, decoder depth, and population seeding. Recovery (%) and mean search time (s) are reported. Bold indicates the highest recovery within each neural policy-depth group.

		Livermore		Nguyen	
Configuration	Trainable params	Recovery ↑	Time (s) ↓	Recovery ↑	Time (s) ↓
Non-LLM population seeding					
Random-seed GP	0	66.91	120.0	91.00	46.8
Single-block policy					
Fully frozen	0	72.91	130.8	90.67	55.2
Interface-only updates	0.027M	72.18	130.2	90.67	54.6
DIGEST	14.94M	**74.00**	118.8	**93.33**	49.2
Full-depth policy					
Interface-only updates	0.027M	73.09	603.0	**93.67**	184.2
LoRA updates	14.87M	**79.27**	1213.8	**93.67**	398.4
Full-dense updates	357.92M	76.36	559.8	**93.67**	184.8

Effect of online adaptation. On Livermore, online adaptation improves recovery at both decoder depths. For the single-block policy, DIGEST reaches 74.00%, compared with 72.91% when fully frozen and 72.18% with interface-only updates. For full-depth policies, recovery rises from 73.09% with interface-only updates to 76.36% with dense updates and 79.27% with LoRA. These results

show that online adaptation is effective, supporting our central idea that allowing the LLM-derived policy to more thoroughly internalize past search discoveries improves subsequent generation. On Nguyen, single-block adaptation improves recovery from 90.67% to 93.33%, while all full-depth variants reach 93.67%, consistent with the more limited headroom on this benchmark.

Effect of Decoder Depth Full-depth adaptation improves recovery at substantially greater search cost. On Livermore, LoRA and dense tuning reach 79.27% and 76.36% recovery in 1213.8 and 559.8 seconds, versus 74.00% in 118.8 seconds for DIGEST. Notably, DIGEST and full-depth LoRA have similar trainable parameter counts (14.94M versus 14.87M), yet very different search times. On Nguyen, full-depth variants improve recovery by only 0.34%. These results support the single-block design as a favorable accuracy-efficiency trade-off.

Effect of Population Seeding On Livermore, recovery increases from 66.91% with random seeding to 72.91% with the frozen single-block policy and 74.00% with DIGEST. The 6.00% gain from random to frozen-LLM seeding shows that, even before online adaptation, the pretrained LLM-derived policy provides a useful structural prior for population generation, supporting the design motivation of leveraging LLM modeling capacity for more flexible candidate generation.

6 Related Work

Conventional symbolic regression. Conventional SR generates candidates through predefined search mechanisms or specialized learned generators. Evolutionary and GP-based methods, evolve expression populations through fitness-based selection and variation (Schmidt & Lipson, 2009; Burlacu et al., 2020; Virgolin et al., 2021; Cranmer, 2023). Other search methods use physics-inspired decomposition (Udrescu & Tegmark, 2020) or Monte Carlo tree search, sometimes with learned guidance (Lu et al., 2021; Sun et al., 2023; Kamienny et al., 2023; Shojaee et al., 2023; Xu et al., 2024; Huang et al., 2026). Specialized generators include DSR, trained with risk-seeking policy gradients (Petersen et al., 2019), and models such as E2E trained on synthetic equation-data pairs (Biggio et al., 2021; Valipour et al., 2021; Kamienny et al., 2022). Hybrid approaches integrate neural generation with GP refinement (Mundhenk et al., 2021; Landajuela et al., 2022), while learned description-length estimates also guide search (Yu et al., 2025).

LLM-based symbolic regression. LLM-based approaches use general-purpose language models to guide expression discovery. ICSR iteratively refines functional forms from numerical observations and previous candidates, with coefficients fitted externally (Merler et al., 2024). LLM-SR generates equation programs from scientific descriptions and reuses high-scoring candidates in subsequent prompts (Shojaee et al., 2025a), while LaSR augments GP with LLM-guided genetic operations and an evolving concept library (Grayeli et al., 2024). Other approaches use simulation feedback (Ma et al., 2024), domain-guided self-evaluation (Song et al., 2025), reflective prompting (Wang et al., 2025), or reusable structure libraries (Li & Pan, 2026). These methods reuse search discoveries through inference-time context, self-evaluation, libraries, or search mechanisms without updating LLM parameters during search. In contrast, DIGEST uses discovered high-quality expressions to update an LLM-derived symbolic policy online, shaping subsequent generation.

7 Conclusion

In this work, we propose DIGEST, which turns GP search discoveries into online training signals for an LLM-derived symbolic policy in symbolic regression. To render more flexible candidate generation, DIGEST repurposes a pretrained decoder block from a lightweight LLM into a symbolic policy that maps expression-tree context to distributions over symbolic tokens for GP population seeding. To more thoroughly utilize search discoveries, high-quality expressions discovered by GP are fed back to update the policy online, allowing it to progressively internalize past discoveries and improve subsequent population generation. Experiments on two ground-truth benchmarks and 122 BlackBox datasets demonstrate strong recovery and predictive performance.

AI USE STATEMENT

We used generative AI tools to assist with editing and polishing the manuscript. All AI-assisted work was reviewed by the authors. The authors take full responsibility for the final content and claims of this work.

REPRODUCIBILITY STATEMENT

Section 3, Section 4 and Algorithm 1 describe the proposed method and experimental setup. Appendix A provides detailed implementation and configuration information, including the symbolic interface, online objective, GP and budget settings, and adaptation controls. Appendix B specifies the benchmark definitions and evaluation protocols, while Appendix C documents baseline configurations and timing details. Appendix D provides detailed experimental results, and Appendix E reproduces the active prompt templates used for the LLM-based baselines. We plan to make the implementation publicly available upon acceptance.

REFERENCES

Daniel A Abolafia, Mohammad Norouzi, Jonathan Shen, Rui Zhao, and Quoc V Le. Neural program synthesis with priority queue training. *arXiv preprint arXiv:1801.03526*, 2018.

Sungsoo Ahn, Junsu Kim, Hankook Lee, and Jinwoo Shin. Guiding deep molecular optimization with genetic exploration. *Advances in neural information processing systems*, 33:12008–12021, 2020.

Luca Biggio, Tommaso Bendinelli, Alexander Neitz, Aurelien Lucchi, and Giambattista Parascandolo. Neural symbolic regression that scales. In *International conference on machine learning*, pp. 936–945. Pmlr, 2021.

Leo Breiman. Random forests. *Mach. Learn.*, 45(1):5–32, 2001. doi: 10.1023/A:1010933404324. URL https://doi.org/10.1023/A:1010933404324.

Bogdan Burlacu, Gabriel Kronberger, and Michael Kommenda. Operon c++ an efficient genetic programming framework for symbolic regression. In *Proceedings of the 2020 genetic and evolutionary computation conference companion*, pp. 1562–1570, 2020.

William G. La Cava, Tilak Raj Singh, James Taggart, Srinivas Suri, and Jason H. Moore. Learning concise representations for regression by evolving networks of trees. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. URL https://openreview.net/forum?id=Hke-JhA9Y7.

Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In Balaji Krishnapuram, Mohak Shah, Alexander J. Smola, Charu C. Aggarwal, Dou Shen, and Rajeev Rastogi (eds.), *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, pp. 785–794. ACM, 2016. doi: 10.1145/2939672.2939785. URL https://doi.org/10.1145/2939672.2939785.

Miles Cranmer. Interpretable machine learning for science with pysr and symbolicregression. jl. *arXiv preprint arXiv:2305.01582*, 2023.

Arya Grayeli, Atharva Sehgal, Omar Costilla-Reyes, Miles Cranmer, and Swarat Chaudhuri. Symbolic regression with a learned concept library. *Advances in Neural Information Processing Systems*, 37:44678–44709, 2024.

Zhengyao Huang, Daniel Huang, Tiannan Xiao, Dina Ma, Zhenyu Ming, Hao Shi, and Yuanhui Wen. Improving monte carlo tree search for symbolic regression. *Advances in Neural Information Processing Systems*, 38:40819–40856, 2026.

Pierre-Alexandre Kamienny, Stéphane d'Ascoli, Guillaume Lample, and François Charton. End-to-end symbolic regression with transformers. *Advances in Neural Information Processing Systems*, 35:10269–10281, 2022.

Pierre-Alexandre Kamienny, Guillaume Lample, Sylvain Lamprier, and Marco Virgolin. Deep generative symbolic regression with monte-carlo-tree-search. In *International Conference on Machine Learning*, pp. 15655–15668. PMLR, 2023.

John R Koza. Genetic programming as a means for programming computers by natural selection. *Statistics and computing*, 4(2):87–112, 1994.

William La Cava, Bogdan Burlacu, Marco Virgolin, Michael Kommenda, Patryk Orzechowski, Fabrício Olivetti de França, Ying Jin, and Jason H Moore. Contemporary symbolic regression methods and their relative performance. *Advances in neural information processing systems*, 2021(DB1):1, 2021.

Mikel Landajuela, Chak Shing Lee, Jiachen Yang, Ruben Glatt, Claudio P Santiago, Ignacio Aravena, Terrell Mundhenk, Garrett Mulcahy, and Brenden K Petersen. A unified framework for deep symbolic regression. *Advances in neural information processing systems*, 35:33985–33998, 2022.

Yunlun Li and Sinno Jialin Pan. Robust equation structure learning with adaptive refinement. In *International Conference on Learning Representations*, volume 2026, pp. 70029–70053, 2026.

Qiang Lu, Fan Tao, Shuo Zhou, and Zhiguang Wang. Incorporating actor-critic in monte carlo tree search for symbolic regression. *Neural Comput. Appl.*, 33(14):8495–8511, 2021. doi: 10.1007/S00521-020-05602-2. URL https://doi.org/10.1007/s00521-020-05602-2.

Pingchuan Ma, Tsun-Hsuan Wang, Minghao Guo, Zhiqing Sun, Joshua B Tenenbaum, Daniela Rus, Chuang Gan, and Wojciech Matusik. Llm and simulation as bilevel optimizers: A new paradigm to advance physical scientific discovery. *arXiv preprint arXiv:2405.09783*, 2024.

Matteo Merler, Katsiaryna Haitsiukevich, Nicola Dainese, and Pekka Marttinen. In-context symbolic regression: Leveraging large language models for function discovery. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 4: Student Research Workshop)*, pp. 427–444, 2024.

T Nathan Mundhenk, Mikel Landajuela, Ruben Glatt, Claudio P Santiago, Daniel M Faissol, and Brenden K Petersen. Symbolic regression via neural-guided genetic programming population seeding. *arXiv preprint arXiv:2111.00053*, 2021.

Brenden K Petersen, Mikel Landajuela, T Nathan Mundhenk, Claudio P Santiago, Soo K Kim, and Joanne T Kim. Deep symbolic regression: Recovering mathematical expressions from data via risk-seeking policy gradients. *arXiv preprint arXiv:1912.04871*, 2019.

Michael Schmidt and Hod Lipson. Distilling free-form natural laws from experimental data. *science*, 324(5923):81–85, 2009.

Parshin Shojaee, Kazem Meidani, Amir Barati Farimani, and Chandan Reddy. Transformer-based planning for symbolic regression. *Advances in Neural Information Processing Systems*, 36: 45907–45919, 2023.

Parshin Shojaee, Kazem Meidani, Shashank Gupta, Amir Barati Farimani, and Chandan K Reddy. Llm-sr: scientific equation discovery via programming with large language models. 2024. *URL: http://arxiv. org/abs/2404.18400*, 2025a.

Parshin Shojaee, Ngoc-Hieu Nguyen, Kazem Meidani, Amir Barati Farimani, Khoa D Doan, and Chandan K Reddy. Llm-srbench: A new benchmark for scientific equation discovery with large language models. *arXiv preprint arXiv:2504.10415*, 2025b.

Zhilong Song, Qionghua Zhou, Chunjin Ren, Chongyi Ling, Minggang Ju, and Jinlan Wang. Llm-feynman: leveraging large language models for universal scientific formula and theory discovery. *arXiv preprint arXiv:2503.06512*, 2025.

Fangzheng Sun, Yang Liu, Jian-Xun Wang, and Hao Sun. Symbolic physics learner: Discovering governing equations via monte carlo tree search, 2023. URL https://arxiv.org/abs/2205.13134.

Llama Team. The llama 3 herd of models. *CoRR*, abs/2407.21783, 2024. doi: 10.48550/ARXIV.2407.21783. URL https://doi.org/10.48550/arXiv.2407.21783.

Silviu-Marian Udrescu and Max Tegmark. Ai feynman: A physics-inspired method for symbolic regression. *Science advances*, 6(16):eaay2631, 2020.

Mojtaba Valipour, Bowen You, Maysum Panju, and Ali Ghodsi. Symbolicgpt: A generative transformer model for symbolic regression. *arXiv preprint arXiv:2106.14131*, 2021.

Marco Virgolin, Tanja Alderliesten, Cees Witteveen, and Peter AN Bosman. Improving model-based genetic programming for symbolic regression of small expressions. *Evolutionary computation*, 29(2):211–237, 2021.

Runxiang Wang, Boxiao Wang, Kai Li, Yifan Zhang, and Jian Cheng. Drsr: Llm based scientific equation discovery with dual reasoning from data and experience. *arXiv preprint arXiv:2506.04282*, 2025.

Yilong Xu, Yang Liu, and Hao Sun. Reinforcement symbolic regression machine. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=PJVUWpPnZC.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report. *CoRR*, abs/2412.15115, 2024. doi: 10.48550/ARXIV.2412.15115. URL https://doi.org/10.48550/arXiv.2412.15115.

Zihan Yu, Jingtao Ding, Yong Li, and Depeng Jin. Symbolic regression via mdlformer-guided search: from minimizing prediction error to minimizing description length. In *International Conference on Learning Representations*, volume 2025, pp. 65306–65333, 2025.

A Implementation Details

This appendix specifies the symbolic interface, search procedure, and reported experimental configurations. The main method uses a single-block policy; full-depth configurations are adaptation controls. AdamW uses separate learning-rate groups for the symbolic interface and decoder parameters, together with weight decay and gradient clipping.

A.1 Symbolic Interface

Canonical labels and initialization. Table 5 gives the fixed token-label mapping used for symbolic embedding initialization. Variables keep generic vocabulary names such as `x1`; unmapped symbols keep their original names. Tokenization adds no special tokens. For a label tokenized into m_a text-token IDs $z_{a,1}, \ldots, z_{a,m_a}$, we initialize its symbolic representation by averaging the corresponding pretrained token embeddings, $\mathbf{e}_a = \frac{1}{m_a} \sum_{j=1}^{m_a} \mathbf{E}_{\mathrm{LM}}[z_{a,j}]$, where $\mathbf{E}_{\mathrm{LM}}$ is the pretrained token-embedding table. No additional vector normalization is applied.

Parent indexing and missing states. Let $\mathcal{A}_{\mathrm{nt}} \subseteq \mathcal{A}$ denote the nonterminal symbols. For every symbol $a \in \mathcal{A}$, the sibling embedding is initialized to $\mathbf{e}_a$ and the corresponding output-projection row to $\mathbf{e}_a^{\mathsf{T}}$. Parent embeddings use the same vectors for $a \in \mathcal{A}_{\mathrm{nt}}$. The parent and sibling tables additionally contain an empty-parent and an empty-sibling entry, respectively. Compact parent indices are mapped to their full symbol IDs during initialization; the embedding notation omits this index conversion. Although both missing states are written as EMPTY for readability, they use different

Table 5: Canonical labels used to initialize the symbolic interface.

Symbol	Text label	Symbol	Text label
`add`	`add`	`sub`	`subtract`
`mul`	`multiply`	`div`	`divide`
`sin`	`sin`	`cos`	`cos`
`tan`	`tan`	`exp`	`exp`
`log`	`log`	`sqrt`	`sqrt`
`n2`	`square`	`inv`	`inverse`
`neg`	`negative`	`const`	`constant`

table entries and initial values:

$$\mathbf{E}_p[\text{EMPTY}] \leftarrow \frac{1}{|\mathcal{A}_{\text{nt}}|} \sum_{a \in \mathcal{A}_{\text{nt}}} \mathbf{e}_a, \tag{4}$$

$$\mathbf{E}_s[\text{EMPTY}] \leftarrow \frac{1}{|\mathcal{A}|} \sum_{a \in \mathcal{A}} \mathbf{e}_a. \tag{5}$$

The parent table, sibling table, and output matrix receive separate copies of their initialization vectors. They are jointly optimized with the policy loss but are not tied to one another. The output bias starts at zero. Output LayerNorm is newly constructed rather than copied from the pretrained model's final normalization.

Structural computation and replay. Parent and sibling embeddings are interleaved and passed to the decoder, with each symbolic step contributing two input positions and prediction performed from the corresponding sibling position. For GP-refined expressions, parent and sibling histories together with the generation constraints are reconstructed from their pre-order sequences before likelihood evaluation. Only valid expression positions contribute to the training loss.

A.2 Feedback Filtering and Online Objective

Feedback batch and reward tail. The generated batch $\mathcal{B}$ is concatenated with at most M expressions from the current GP run's archive to form $\mathcal{C}$. The archive return is not the full final population. No deduplication is performed at this stage, so repeated expressions retain their multiplicity in reward quantiles and entropy averages.

The empirical $(1-\epsilon)$ reward quantile used to form $\mathcal{C}_\epsilon$ follows the `higher` convention. Every candidate tied at the threshold is kept; therefore $|\mathcal{C}_\epsilon|$ need not equal $\epsilon|\mathcal{C}|$. Filtering uses the evaluator-provided rewards, without separately removing trajectories by their invalid-status flags.

Queue admission and sampling. Only the highest-reward member of $\mathcal{C}_\epsilon$ is submitted to $\mathcal{Q}$. If several candidates tie for this nomination, the first occurrence in the batch is selected. Queue uniqueness uses `program.str`, a byte-string encoding of the expression's token-ID sequence, not a readable expression string or a symbolic-equivalence test. Fitted coefficient values are not part of this key. A nominee already present is ignored; the procedure does not substitute the next-best candidate. The queue keeps up to K high-reward distinct submitted expressions, not necessarily the best K among all expressions ever evaluated. Each update samples one queue entry uniformly; the reported queue capacity is $K = 10$.

Loss reduction. The sequence negative log-likelihood and entropy are

$$-\log \pi_\theta(\phi) = -\sum_{t=1}^{|\phi|} \log \pi_\theta(\phi_t \mid \phi_{<t}), \tag{6}$$

$$H_\theta(\phi) = \sum_{t=1}^{|\phi|} H(\pi_\theta(\cdot \mid \phi_{<t}))\,. \tag{7}$$

These expressions refer to valid positions retained by the loss mask. For $H(\mathbf{p}) = -\sum_a p_a \log p_a$, we use $0 \log 0 = 0$. The positional entropy-discount factor is one. Negative log-likelihood uses the sampled queue trajectory; entropy is averaged over the current filtered feedback batch, which may contain both policy-generated and GP-refined expressions. Neither sequence quantity is divided by its length.

A baseline value computed by the training routine is not used by this objective. For a fixed candidate batch with finite rewards, the highest-reward member survives reward-tail filtering. Consequently, changing ϵ does not change that batch's queue nominee; its direct effect is on the trajectories used for entropy regularization. This can change later updates and search outcomes. The filter is not an additional risk-seeking policy-gradient objective.

A.3 GP, Evaluation, and Budget Accounting

Training reward. For numerically valid predictions, the training reward is

$$\begin{aligned} \mathrm{NRMSE}(\phi) &= \frac{1}{\sigma_y}\sqrt{\frac{1}{n}\sum_{i=1}^{n}\big(y_i - f_\phi(\mathbf{x}_i)\big)^2}, \\ R(\phi) &= \frac{1}{1+\mathrm{NRMSE}(\phi)}. \end{aligned} \tag{8}$$

Here σ_y is the standard deviation of the training targets. Free coefficients, when present, are fitted to the training observations before fitness evaluation. This reward is separate from the test-set recovery criterion.

The recovery criterion for Nguyen and Livermore is specified in Appendix B.1.

Genetic operations. GP uses tournament selection and one-point subtree crossover. Mutation chooses uniformly among subtree replacement, node replacement, insertion, and shrink; no custom `mutation_probs` are supplied. If a decorated genetic operation violates a configured constraint, the offspring is replaced by a copy of a randomly selected input parent. The full generated batch initializes each GP run. The seeded DEAP individuals initially have unset fitness. Fitness values are assigned when offspring are evaluated, reusing cached rewards where available; there is no separate pass that assigns fitness to every initial seed before the first tournament. After GP refinement, the generated batch and archive returns are concatenated and their training rewards are obtained, reusing cached evaluations. The GP population and per-run archive reset at each restart; policy parameters, AdamW moment estimates, the priority queue, expression cache, and best-so-far expression persist. The optimizer is constructed once, and the cache is cleared once at task setup.

Budget and stopping. The evaluation counter increases by N for policy samples and by the GP run's `nevals`. The latter counts individuals submitted for fitness evaluation and can include cache hits. The budget is not a count of unique uncached expressions or all numerical calls inside coefficient fitting. For nominal budget B, the loop permits at most $\lfloor B/N \rfloor$ restarts and checks $b > B$ after queue processing and the online update. The final restart may therefore exceed B. Configured stopping conditions are checked at the same restart boundary. The search loop's best-so-far expression is updated from the combined feedback batch. The additional SRBench time limit and its final reporting selection are specified in Appendix A.4.

Runtime. The internal timer measures search and online adaptation. It starts after task and controller initialization, excluding model loading and other initialization costs.

A.4 Main Experimental Configuration

Pretrained source and decoder configuration. The reported experiments instantiate the lightweight pretrained source with Qwen2.5-0.5B (Yang et al., 2024). The main policy executes its decoder block at zero-based index 10, namely the eleventh block. Other decoder blocks are not executed. The original text embedding lookup is frozen and bypassed; the original final normalization is bypassed in favor of the trainable symbolic output LayerNorm. The decoder block, parent and sibling embeddings, output LayerNorm, projection, and output bias are all trained.

In the one-variable, eight-operator Nguyen configuration, the main policy has $14{,}939{,}273$ trainable parameters. With the two-variable Nguyen vocabulary it has $14{,}941{,}066$. These counts include the symbolic interface and exclude frozen parameters. Each added variable introduces a sibling-embedding row, an output-projection row, and a bias entry; therefore 14.94M is a representative count, not a vocabulary-independent size.

Shared settings. The generated population has $N = 500$ expressions. Each GP run returns at most $M = 1$ archive expression. Crossover and mutation probabilities are both 0.5, the tournament size is five, and the maximum generated mutation-subtree depth is three. The queue capacity is $K = 10$, the reward-tail parameter is $\epsilon = 0.02$, and each restart uses one queue sample and one AdamW step. Sampling temperature is one. Weight decay is 0.01, global gradient-norm clipping is 1.0, and the symbolic-interface learning rate is 10^{-5}.

Vocabulary and constraints. The operator set is $\{+, -, \times, \div, \sin, \cos, \exp, \log\}$ for all three benchmark suites. Nguyen and Livermore use input-variable terminals without free-constant tokens; SRBench additionally includes the free-constant placeholder `const`. These benchmark vocabularies are distinct from the full set of supported initialization labels in Table 5.

For Nguyen and Livermore, the active hard constraints enforce lengths from 4 to 30, prohibit immediately composed inverse unary operators ($\exp$ and $\log$), and prohibit nested trigonometric operators. The configuration also enables a constant-repeat limit of 10, the constant-child constraint, and the input-presence constraint; these are skipped as inapplicable for the constant-free vocabulary.

For SRBench, the length range is 4 to 60 and at most 20 `const` tokens are permitted. Inverse-unary and nested-trigonometric constraints remain active. The `const` token cannot be the only unique child of an operator, and every expression must contain an input variable. The generic relational, uniform-arity, soft-length, and language-model priors are disabled in all three settings.

Nguyen and Livermore. Both use a nominal budget of 2×10^6 expression evaluations, $S = 25$ GP generations per restart, and maximum expression length 30. The decoder learning rate is 10^{-6} on Nguyen and 3×10^{-6} on Livermore. PQT weights are 100 and 200, respectively; entropy weights are 0.001 and 0.005. Neither vocabulary includes a free-constant token. Recovery-based early stopping is enabled.

SRBench BlackBox. The nominal budget is 5×10^5, with $S = 50$, maximum expression length 60, decoder learning rate 10^{-6}, PQT weight 100, and entropy weight 0.005. Expressions may contain up to 20 free constants.

For final reporting, the SRBench wrapper selects the cached program with the highest training reward and evaluates it on the test data. No additional final coefficient refit is requested in the reported configuration. The wrapper's final reporting steps are outside the internal search timer.

A.5 Full-Depth Adaptation Controls

These controls are ablations of decoder depth and update parameterization, not additional stages of the main method. Both execute all 24 decoder blocks and densely train the symbolic interface. Their original text embedding table remains frozen and bypassed, and the original final normalization is replaced by the new output LayerNorm.

Full-model LoRA. Base decoder weights and biases are frozen, while low-rank updates are trained for `q_proj`, `k_proj`, `v_proj`, `o_proj`, `gate_proj`, `up_proj`, and `down_proj`. The rank is 27, dropout is zero, and the scaling parameter is 27 on Nguyen and 54 on Livermore. The scaling parameter is not an optimizer learning rate. LoRA uses the decoder learning-rate group, without separate attention or feed-forward learning-rate overrides. The Nguyen configuration uses PEFT's default initializer and an empty `alpha_pattern`, without module-specific scaling overrides.

With the one-variable, eight-operator Nguyen vocabulary, this control has $14{,}873{,}865$ trainable parameters including the symbolic interface.

Full-model dense tuning. All decoder blocks and the symbolic interface are densely updated.

The corresponding one-variable Nguyen configuration has $357{,}924{,}105$ trainable parameters. Here, full model refers to the complete decoder stack, not all original text-language-model parameters. As in the main policy, parameter counts depend on the symbolic vocabulary.

B Benchmarks and Evaluation Protocols

B.1 Ground-Truth Benchmarks

Benchmark setup. We follow the Nguyen and Livermore benchmark definitions of Mundhenk et al. (2021). Our Nguyen suite contains Nguyen-1–11 and Nguyen-12*, where Nguyen-12* uses inputs sampled from $[0, 10]$ rather than $[0, 1]$; Livermore contains 22 equations. We write $U(a, b, n)$ for n points sampled uniformly from $[a, b]$ for each input variable. For each equation, the training and test sets are sampled separately from the prescribed domain and then held fixed across all random seeds. For Nguyen and Livermore, our evaluations use 25 independent random seeds per equation. Tables 6 and 7 list the target expressions and sampling domains.

Table 6: Target expressions and sampling domains for the Nguyen benchmark used in our experiments.

Problem	Target expression	Sampling domain
Nguyen-1	$x^3 + x^2 + x$	$U(-1, 1, 20)$
Nguyen-2	$x^4 + x^3 + x^2 + x$	$U(-1, 1, 20)$
Nguyen-3	$x^5 + x^4 + x^3 + x^2 + x$	$U(-1, 1, 20)$
Nguyen-4	$x^6 + x^5 + x^4 + x^3 + x^2 + x$	$U(-1, 1, 20)$
Nguyen-5	$\sin(x^2)\cos(x) - 1$	$U(-1, 1, 20)$
Nguyen-6	$\sin(x) + \sin(x + x^2)$	$U(-1, 1, 20)$
Nguyen-7	$\log(x + 1) + \log(x^2 + 1)$	$U(0, 2, 20)$
Nguyen-8	$\sqrt{x}$	$U(0, 4, 20)$
Nguyen-9	$\sin(x) + \sin(y^2)$	$U(0, 1, 20)$
Nguyen-10	$2\sin(x)\cos(y)$	$U(0, 1, 20)$
Nguyen-11	x^y	$U(0, 1, 20)$
Nguyen-12*	$x^4 - x^3 + \frac{1}{2}y^2 - y$	$U(0, 10, 20)$

Recovery criterion. For our Nguyen and Livermore experiments, we follow the numerical success criterion in the released NGGP/DSO implementation Mundhenk et al. (2021): a run is counted as recovered if its normalized mean squared error on the noiseless test set satisfies $\mathrm{NMSE}_{\mathrm{test}} < 10^{-12}$. Recovery rate is the percentage of runs satisfying this criterion. We apply the same criterion to all methods evaluated in our experiments. For aggregate benchmark results, we average the recovery rates across equations. For the LLM-based comparison, we additionally report median test NMSE and mean search time. Timing details are provided in Appendix C.3.

B.2 SRBench BlackBox

We evaluate on the 122 SRBench BlackBox datasets (La Cava et al., 2021). Each run uses a seed-specific 75/25 train/test split, without input or target scaling. We perform 10 independent runs per dataset with a 5×10^5 evaluation budget per run and report cross-dataset median test R^2 and median model size. A Pareto view of the accuracy-complexity trade-off, where DIGEST lies on the Pareto frontier, is provided in Appendix D.2.

B.3 TPSR Evaluation Setting

TPSR (Shojaee et al., 2023) uses an E2E Transformer pretrained for at most ten input dimensions. Its official BlackBox setting is described as 57 continuous-feature datasets with at most ten inputs. Table 12 reports our separate comparison under this regime, using median test R^2 and mean model size. These results are reported separately from the 122-task median-size analysis.

Table 7: Target expressions and sampling domains for the Livermore benchmark used in our experiments.

Problem	Target expression	Sampling domain
Livermore-1	$\frac{1}{3} + x + \sin(x^2)$	$U(-10, 10, 1000)$
Livermore-2	$\sin(x^2)\cos(x) - 2$	$U(-1, 1, 20)$
Livermore-3	$\sin(x^3)\cos(x^2) - 1$	$U(-1, 1, 20)$
Livermore-4	$\log(x+1) + \log(x^2+1) + \log(x)$	$U(0, 2, 20)$
Livermore-5	$x^4 - x^3 + x^2 - y$	$U(0, 1, 20)$
Livermore-6	$4x^4 + 3x^3 + 2x^2 + x$	$U(-1, 1, 20)$
Livermore-7	$\sinh(x)$	$U(-1, 1, 20)$
Livermore-8	$\cosh(x)$	$U(-1, 1, 20)$
Livermore-9	$x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x$	$U(-1, 1, 20)$
Livermore-10	$6\sin(x)\cos(y)$	$U(0, 1, 20)$
Livermore-11	$\frac{x^2 y^2}{x+y}$	$U(-1, 1, 50)$
Livermore-12	$\frac{x^5}{y^3}$	$U(-1, 1, 50)$
Livermore-13	$x^{1/3}$	$U(0, 4, 20)$
Livermore-14	$x^3 + x^2 + x + \sin(x) + \sin(x^2)$	$U(-1, 1, 20)$
Livermore-15	$x^{1/5}$	$U(0, 4, 20)$
Livermore-16	$x^{2/5}$	$U(0, 4, 20)$
Livermore-17	$4\sin(x)\cos(y)$	$U(0, 1, 20)$
Livermore-18	$\sin(x^2)\cos(x) - 5$	$U(-1, 1, 20)$
Livermore-19	$x^5 + x^4 + x^2 + x$	$U(-1, 1, 20)$
Livermore-20	$\exp(-x^2)$	$U(-1, 1, 20)$
Livermore-21	$x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x$	$U(-1, 1, 20)$
Livermore-22	$\exp(-0.5x^2)$	$U(-1, 1, 20)$

C Baseline Methods and Implementations

C.1 Conventional Baselines

The configurations for DSR, GEGL, and NGGP follow the parameterizations reported by Mundhenk et al. (2021). PySR and iMCTS follow the configurations reported by Huang et al. (2026).

DSR. Deep Symbolic Regression uses a recurrent neural network to define a probability distribution over mathematical expressions. A risk-seeking policy-gradient objective trains the generator to favor high-reward expressions (Petersen et al., 2019).

GEGL. Genetic Expert-Guided Learning couples a neural generator with a genetic expert that produces high-quality candidates for imitation learning (Ahn et al., 2020). The symbolic-regression comparison uses the configuration described by Mundhenk et al. (2021).

NGGP. Neural-Guided Genetic Programming combines a neural expression generator with GP searches. Expressions obtained from generation and GP refinement provide feedback for updating the generator (Mundhenk et al., 2021).

PySR. PySR evolves expression populations through fitness-based selection and genetic variation (Cranmer, 2023).

iMCTS. iMCTS augments Monte Carlo tree search with an extreme-bandit allocation strategy and evolution-inspired state jumps, including mutation and crossover, to explore nonlocal changes to candidate expressions (Huang et al., 2026).

SRBench reference methods. The BlackBox comparison includes the 21 baseline methods reported by La Cava et al. (2021). These comprise symbolic regressors such as Operon, FEAT (Cava

et al., 2019), and GP-GOMEA, together with standard machine-learning regressors such as XGBoost (Chen & Guestrin, 2016) and Random Forest (Breiman, 2001).

SR4MDL and TPSR. SR4MDL guides symbolic search with learned description-length estimates (Yu et al., 2025). TPSR incorporates Monte Carlo tree search into Transformer-based expression generation (Shojaee et al., 2023). The separate TPSR evaluation setting is described in Appendix B.3.

C.2 LLM-Based Baselines

We evaluate ICSR (Merler et al., 2024), LLM-SR (Shojaee et al., 2025a), LaSR (Grayeli et al., 2024), and RESTART (Li & Pan, 2026) using their released implementations. We retain the default configurations of the released implementations except for the context and budget changes described below. Key search and inference parameters are reported for reproducibility. We make two main changes for our evaluation. First, to match the no-scientific-context setting of Nguyen and Livermore, we replace task-specific scientific descriptions with neutral input and target descriptions where applicable, while retaining generic mathematical instructions and each method's native search-feedback mechanism. Second, to keep search costs in a comparable range to DIGEST, we use bounded method-specific search budgets: LLM-SR is capped at 100 generated programs, LaSR uses 20 GP iterations with 20 cycles per iteration, and RESTART is capped at 20 generated candidates; ICSR retains its default ten refinement iterations. Complete prompt templates are provided in Appendix E.

Shared backend for LLM-SR and LaSR. LLM-SR and LaSR are evaluated through the public LLM-SRBench implementation (Shojaee et al., 2025b), using its default configurations for the two methods except for the context and budget changes described above. Both use Llama-3.1-8B-Instruct (Team, 2024).

C.2.1 ICSR

ICSR uses Llama-3-8B-Instruct (Team, 2024). Its released text-only prompts are generic and remain unchanged.

ICSR settings. The search uses ten refinement iterations and up to five previous functions with their training scores. Temperature is 1.0, top-p is 0.90, and top-k is 60. The output limits are 512 tokens for seed generation and 256 tokens for refinement. Native retries, early stopping, and the reference-equation stopping check are retained.

Prompt construction. Each prompt contains up to 40 training points selected by the released implementation, together with previous functions and their training scores during refinement. Displayed coefficients are replaced by c. The complete seed-generation and refinement templates are provided in Appendix E.1.

C.2.2 LLM-SR

LLM-SR settings. The search uses four candidates per prompt, ten islands, and up to two prior program examples. In our evaluation, we cap the search at 100 generated programs. Candidate execution uses a 30-second timeout, and SciPy BFGS is used for coefficient optimization. Exact-zero-training-error stopping is retained.

Prompt construction. Each request combines a neutral task instruction with the program prompt. The initial prompt uses an affine seed program, while later requests include up to two previously discovered programs as context. The complete one- and two-input prompt templates are provided in Appendix E.2.

C.2.3 LaSR

LaSR settings. The search uses ten populations of size 33 and a maximum expression size of 30. In our evaluation, we use 20 GP iterations with 20 cycles per iteration. The weights for LLM-based

mutation, crossover, and random generation are 0.005 each. The recent-concept window is 20 and `num_pareto_context=3`. The operator set includes `tan` and `cosh`, while the existing nesting constraints are retained. All LLM-assisted operations and native concept extraction and evolution remain enabled, and each run starts with a fresh concept library.

Prompts and context adaptation. To remove task-specific scientific context, we use a neutral task description as the first suggestion for generation, mutation, and crossover, and remove the physical-concept clause from `extract_idea_system.txt`. The remaining active prompt templates are unchanged. The complete templates, neutral task descriptions, and operator list are provided in Appendix E.3.

C.2.4 RESTART

RESTART settings. RESTART uses Qwen3-8B with thinking disabled. The search uses ten islands, up to two prior program examples, and four completions per batch. In our evaluation, we cap the search at 20 generated candidates. Invalid generated candidates count toward this cap, while initial E2E expressions and conditional concept requests are counted separately. Candidate requests use `max_tokens=30000`; concept requests use temperature 0.7 and `max_tokens=5096`.

RESTART uses the released E2E model for its native residual feedback and retains its concept-library mechanism. The search timer and runtime boundary are described in Appendix C.3.

Prompts and context adaptation. To remove task-specific scientific context, we replace target and input descriptions with neutral names and use mathematical rather than scientific instructions. Only training data are used during discovery. Candidate prompts may include previous programs, E2E feedback consisting of the current candidate's training error and boosted expression, and selected library concepts. Concept-improvement requests use the improvement between previous and new candidates to propose new mathematical concepts. The complete prompt templates and neutral-context substitutions are provided in Appendix E.4.

C.3 Computation and Timing

All locally evaluated methods use one NVIDIA A40 GPU per run on the NCSA Delta cluster. Reported search time excludes initial LLM loading and is averaged over all runs.

DIGEST. The internal timer and reporting boundary of DIGEST are specified in Appendices A.3 and A.4.

LLM-SR and LaSR. LLM-SR and LaSR measure time around `discover()`, including final training-data coefficient fitting and any first-fit Julia initialization inside the call. Model-service startup and subsequent test evaluation are outside this interval.

RESTART. The search timer includes E2E initialization, reloads during discovery, and final training fits, but excludes initial LLM service startup.

D Detailed Experimental Results and Analyses

D.1 Per-Equation Ground-Truth Recovery

Tables 8–11 report results on all 34 equations. Benchmark definitions and evaluation protocols are provided in Appendix B.1, and baseline configurations are provided in Appendix C. Frozen and interface-only controls are summarized in Table 4.

Table 8: Per-equation recovery (%) on Nguyen: conventional baselines and DIGEST. The best value per row is bold.

Equation	DSR	GEGL	NGGP	PySR	iMCTS	DIGEST
Nguyen-1	**100**	**100**	**100**	**100**	**100**	**100.00**
Nguyen-2	**100**	**100**	**100**	98	**100**	**100.00**
Nguyen-3	**100**	**100**	**100**	70	**100**	**100.00**
Nguyen-4	**100**	**100**	**100**	2	97	**100.00**
Nguyen-5	72	92	**100**	89	**100**	**100.00**
Nguyen-6	**100**	**100**	**100**	**100**	**100**	**100.00**
Nguyen-7	35	48	96	35	**100**	80.00
Nguyen-8	96	**100**	**100**	99	**100**	**100.00**
Nguyen-9	**100**	**100**	**100**	**100**	**100**	**100.00**
Nguyen-10	**100**	92	**100**	**100**	**100**	**100.00**
Nguyen-11	**100**	**100**	**100**	**100**	**100**	**100.00**
Nguyen-12*	0	0	12	0	22	**40.00**
Average	83.58	86.00	92.33	74.41	93.25	**93.33**

Table 9: Per-equation recovery (%) on Nguyen: LLM-based baselines, Random-seed GP, DIGEST, and full-depth ablations. The best value per row is bold.

Equation	ICSR	LLM-SR	LaSR	RESTART	Random GP	DIGEST	Full LoRA	Full dense
Nguyen-1	20.00	16.00	28.00	24.00	**100.00**	**100.00**	**100.00**	**100.00**
Nguyen-2	4.00	0.00	4.00	8.00	**100.00**	**100.00**	**100.00**	**100.00**
Nguyen-3	0.00	12.00	0.00	0.00	**100.00**	**100.00**	**100.00**	**100.00**
Nguyen-4	0.00	0.00	0.00	4.00	**100.00**	**100.00**	**100.00**	**100.00**
Nguyen-5	0.00	0.00	0.00	0.00	**100.00**	**100.00**	**100.00**	**100.00**
Nguyen-6	0.00	0.00	32.00	0.00	**100.00**	**100.00**	**100.00**	**100.00**
Nguyen-7	0.00	0.00	0.00	0.00	88.00	80.00	**100.00**	**100.00**
Nguyen-8	48.00	0.00	**100.00**	12.00	**100.00**	**100.00**	**100.00**	**100.00**
Nguyen-9	0.00	0.00	80.00	0.00	**100.00**	**100.00**	**100.00**	**100.00**
Nguyen-10	56.00	0.00	12.00	0.00	**100.00**	**100.00**	**100.00**	**100.00**
Nguyen-11	40.00	0.00	**100.00**	0.00	**100.00**	**100.00**	**100.00**	**100.00**
Nguyen-12*	0.00	12.00	0.00	0.00	4.00	**40.00**	24.00	24.00
Average	14.00	3.33	29.67	4.00	91.00	93.33	**93.67**	**93.67**

Table 10: Per-equation recovery (%) on Livermore: conventional baselines and DIGEST. The best value per row is bold.

Equation	DSR	GEGL	NGGP	PySR	iMCTS	DIGEST
Livermore-1	3	**100**	**100**	**100**	**100**	**100.00**
Livermore-2	87	44	**100**	32	95	**100.00**
Livermore-3	66	**100**	**100**	97	**100**	**100.00**
Livermore-4	76	**100**	**100**	90	**100**	**100.00**
Livermore-5	0	0	4	30	**78**	16.00
Livermore-6	**97**	64	88	0	19	48.00
Livermore-7	0	0	0	0	**15**	0.00
Livermore-8	0	0	0	0	**22**	0.00
Livermore-9	0	12	**24**	0	2	20.00
Livermore-10	0	0	24	26	**35**	16.00
Livermore-11	17	92	**100**	92	**100**	**100.00**
Livermore-12	61	**100**	**100**	18	91	**100.00**
Livermore-13	55	84	**100**	**100**	**100**	**100.00**
Livermore-14	0	**100**	**100**	15	**100**	**100.00**
Livermore-15	0	96	**100**	74	**100**	**100.00**
Livermore-16	4	12	92	**93**	72	80.00
Livermore-17	0	4	68	72	69	**100.00**
Livermore-18	0	0	56	3	29	**100.00**
Livermore-19	**100**	**100**	**100**	88	**100**	**100.00**
Livermore-20	98	**100**	**100**	76	**100**	**100.00**
Livermore-21	2	64	24	0	41	**100.00**
Livermore-22	3	68	84	9	**100**	48.00
Average	30.41	56.36	71.09	46.14	71.41	**74.00**

Table 11: Per-equation recovery (%) on Livermore: LLM-based baselines, Random-seed GP, DIGEST, and full-depth ablations. The best value per row is bold.

Equation	ICSR	LLM-SR	LaSR	RESTART	Random GP	DIGEST	Full LoRA	Full dense
Livermore-1	0.00	0.00	96.00	0.00	**100.00**	**100.00**	**100.00**	**100.00**
Livermore-2	0.00	0.00	0.00	0.00	**100.00**	**100.00**	**100.00**	**100.00**
Livermore-3	0.00	0.00	0.00	0.00	92.00	**100.00**	**100.00**	**100.00**
Livermore-4	0.00	0.00	0.00	0.00	**100.00**	**100.00**	**100.00**	**100.00**
Livermore-5	0.00	0.00	0.00	0.00	4.00	16.00	**52.00**	36.00
Livermore-6	0.00	28.00	0.00	36.00	4.00	48.00	**76.00**	68.00
Livermore-7	**76.00**	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Livermore-8	56.00	0.00	**100.00**	0.00	0.00	0.00	0.00	0.00
Livermore-9	0.00	0.00	0.00	0.00	12.00	20.00	**84.00**	64.00
Livermore-10	**56.00**	0.00	12.00	0.00	8.00	16.00	44.00	20.00
Livermore-11	0.00	0.00	0.00	0.00	**100.00**	**100.00**	**100.00**	**100.00**
Livermore-12	0.00	0.00	0.00	0.00	**100.00**	**100.00**	**100.00**	**100.00**
Livermore-13	0.00	0.00	56.00	0.00	**100.00**	**100.00**	**100.00**	**100.00**
Livermore-14	0.00	0.00	0.00	0.00	**100.00**	**100.00**	**100.00**	**100.00**
Livermore-15	60.00	0.00	60.00	0.00	**100.00**	**100.00**	**100.00**	**100.00**
Livermore-16	0.00	0.00	56.00	0.00	**80.00**	**80.00**	40.00	52.00
Livermore-17	76.00	0.00	4.00	0.00	60.00	**100.00**	96.00	96.00
Livermore-18	0.00	0.00	0.00	0.00	36.00	**100.00**	**100.00**	**100.00**
Livermore-19	0.00	0.00	4.00	12.00	**100.00**	**100.00**	**100.00**	**100.00**
Livermore-20	32.00	0.00	16.00	0.00	**100.00**	**100.00**	**100.00**	**100.00**
Livermore-21	0.00	0.00	0.00	0.00	96.00	**100.00**	**100.00**	**100.00**
Livermore-22	12.00	0.00	16.00	0.00	**80.00**	48.00	52.00	44.00
Average	16.73	1.27	19.09	2.18	66.91	74.00	**79.27**	76.36

D.2 Accuracy–Complexity Analysis on SRBench BlackBox

Pareto view. Figure 5 reports cross-dataset median test R^2 and median model size separately, while Figure 6 shows their joint accuracy-complexity trade-off. DIGEST lies on the Pareto frontier, achieving the highest median test R^2 among the compared methods while maintaining moderate model size.

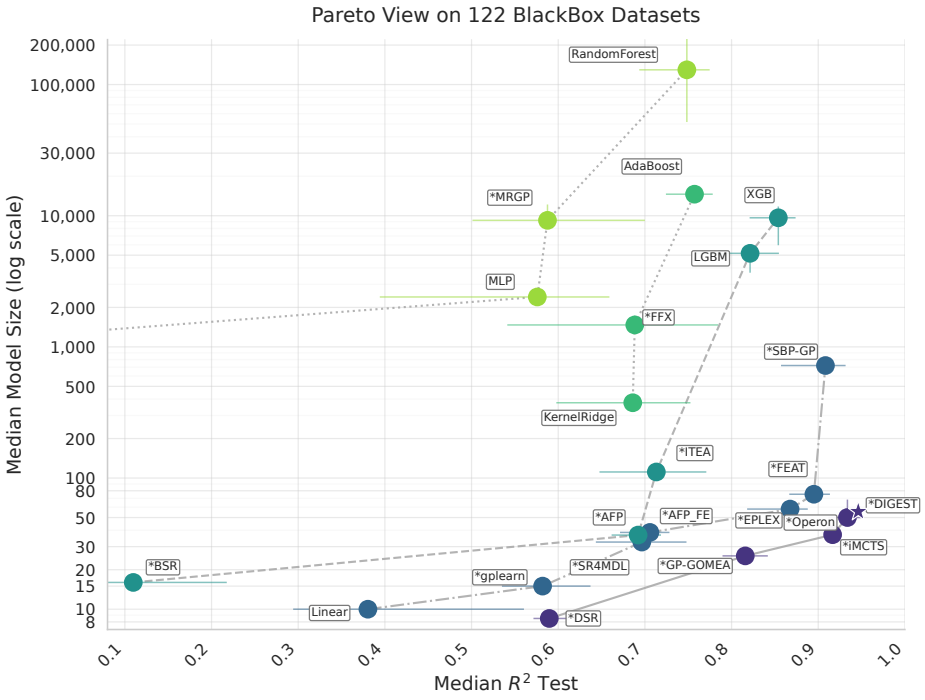

Figure 6: Pareto view of cross-dataset median test R^2 and median model size on 122 SRBench BlackBox datasets. DIGEST lies on the Pareto frontier with the highest median test R^2 and moderate model size. The model-size axis is logarithmic; asterisks identify symbolic regression methods.

TPSR comparison. Table 12 reports the separate comparison under the setting described in Appendix B.3, using median test R^2 and mean model size.

Table 12: Comparison under the TPSR BlackBox evaluation setting.

Method	Median test R^2 ↑	Mean model size ↓
TPSR	0.945	95.71
DIGEST	**0.956**	**53.44**

E Active Prompt Templates

The templates below are reproduced from the active implementations. Dynamic expressions and feedback are filled from the training search. Baseline implementation details are provided in Appendix C.2; benchmark definitions and evaluation protocols are provided in Appendix B.

E.1 ICSR Prompts

Message construction and substitutions are described in Appendix C.2.1.

Seed-generation template (complete).

```
I want you to act as a mathematical function generator.
Given a set of points below, you are to come up with 5 potential functions that would fit the
points. Don't worry too much about accuracy: your task is to generate a set of functions that
are as diverse as possible, so that they can serve as starting points for further
optimization.

To generate the functions, you will start from a set of basic operators and expressions, and
combine them into something more complex.
Your options are:
- {num_variables} independent variable symbols: {variables_list}.
- A coefficient symbol: c (there is no need to write a number - write this generic coefficient
instead).
- Basic operators: +, -, *, /, ^, sqrt, exp, log, abs
- Trigonometric expressions: sin, cos, tan, sinh, cosh, tanh
- Standard constants: "pi" represents pi and "E" represents euler's constant.

Make sure there are no numbers in the functions, use the coefficient token 'c' instead.
You are required to use at least one of each variables from the available list
({variables_list}).
Analyze the points carefully: if there are any negative points in the input, sqrt and log can
not be used unless the input can never be negative. Be careful about dividing by zero!
The functions should all begin with the indicators "f1 = ", "f2 = "... Only write the new
function and no additional output.
Your task is to combine an arbitrary number of these basic blocks to create a complex
expression. Don't be afraid to be creative and experiment! The functions should be as complex
as possible, combining many different operations. Variety is key!

Points: {points}

Functions:
```

Refinement template (complete).

```
I want you to act as a mathematical function generator.
You are given a set of points with (x, y) coordinates below:
{points}
Below are some previous functions and the error they make on the points above. The errors are
arranged in order of their fit values, with the highest values coming first, and lower is
better.

Your task is to give me a list of five new functions that are different from all the ones
reported below, and have a lower error value than all of the ones below. Only output the new
functions and nothing else.
Remember that the functions you generate should always have at most {num_variables} variables
{variables_list}.
The functions should have parametric form, using 'c' in place of any constant or coefficient.
The coefficients will be optimized to fit the data. Make absolutely sure that the functions
you generate are completely different from the ones already given to you.
Remember that you can combine the simple building blocks (operations, constants, variables) in
any way you want to generate more complex functions. Don't be afraid to experiment!
The functions should all begin with the indicators "f1(x) = ", "f2(x) = "...

{functions}
```

E.2 LLM-SR PROMPTS

Message construction is described in Appendix C.2.2.

Complete initial user message, one input.

```
You are a helpful assistant tasked with discovering mathematical function structures from
numerical data. Complete the 'equation' function below, considering the mathematical
relationships among the inputs.


"""
Find the mathematical function skeleton that represents target value, given data on input
variable x1.
"""


import numpy as np

#Initialize parameters
MAX_NPARAMS = 10
params = [1.0]*MAX_NPARAMS
```

```
def equation_v0(x1: np.ndarray, params: np.ndarray) -> np.ndarray:
    """ Mathematical function for target value

    Args:
        x1: A numpy array representing observations of input variable x1.
        params: Array of numeric constants or parameters to be optimized

    Return:
        A numpy array representing target value as the result of applying the mathematical
        function to the inputs.
    """
    output = params[0] * x1 + params[1]
    return output


def equation_v1(x1: np.ndarray, params: np.ndarray) -> np.ndarray:
    """Improved version of `equation_v0`."""
```

Complete initial user message, two inputs.

```
You are a helpful assistant tasked with discovering mathematical function structures from
numerical data. Complete the 'equation' function below, considering the mathematical
relationships among the inputs.


"""
Find the mathematical function skeleton that represents target value, given data on input
variable x1, and input variable x2.
"""


import numpy as np

#Initialize parameters
MAX_NPARAMS = 10
params = [1.0]*MAX_NPARAMS


def equation_v0(x1: np.ndarray, x2: np.ndarray, params: np.ndarray) -> np.ndarray:
    """ Mathematical function for target value

    Args:
        x1: A numpy array representing observations of input variable x1.
        x2: A numpy array representing observations of input variable x2.
        params: Array of numeric constants or parameters to be optimized

    Return:
        A numpy array representing target value as the result of applying the mathematical
        function to the inputs.
    """
    output = params[0] * x1 + params[1] * x2 + params[2]
    return output


def equation_v1(x1: np.ndarray, x2: np.ndarray, params: np.ndarray) -> np.ndarray:
    """Improved version of `equation_v0`."""
```

E.3 LASR PROMPTS

Template substitutions and message roles are described in Appendix C.2.3.

Neutral task context, one input.

```
The expression represents target value (y), given data on input variable x1 (x1).
```

Neutral task context, two inputs.

```
The expression represents target value (y), given data on input variable x1 (x1), and input
variable x2 (x2).
```

Available operators.

```
binary operators: +, *, -, /, ^, and unary operators: exp, log, sqrt, sin, cos, tan, cosh
```

gen_random_system.txt.

```
You are a helpful assistant that proposes a mathematical expression by following three
provided suggestions.
An expression must consist of the following variables: {{variables}}. All constants will be
represented with the symbol C. Each expression will only use these operators: {{operators}}.
```

gen_random_user.txt.

```
Suggestion 1: {{assump1}}
Suggestion 2: {{assump2}}
Suggestion 3: {{assump3}}

Propose {{N}} expressions that would be appropriate given the suggestions. Provide short
commentary for each of your decisions. End with a JSON list that enumerates the proposed
expressions following this format:
```json
["expr1",
 "expr2",
 ...
 "expr{{N}}"
]
```
```

mutate_system.txt.

```
You are a helpful assistant that mutates a mathematical expression by following a few provided
suggestions. You will be given three suggestions and a single reference expression to mutate.
An expression must consist of the following variables: {{variables}}. All constants will be
represented with the symbol C. Each expression will only use these operators: {{operators}}.
```

mutate_user.txt.

```
Suggestion 1: {{assump1}}
Suggestion 2: {{assump2}}
Suggestion 3: {{assump3}}
Reference Expression: {{expr}}

Propose {{N}} expressions that would be appropriate given the suggestions and references.
Provide short commentary for each of your decisions. End with a JSON list that enumerates the
proposed expressions following this format:
```json
["expr1",
 "expr2",
 ...
 "expr{{N}}"
]
```
```

crossover_system.txt.

```
You are a helpful assistant that recombines two mathematical expressions by following a few
provided suggestions. You will be given three suggestions and two reference expressions to
recombine.
An expression must consist of the following variables: {{variables}}. All constants will be
represented with the symbol C. Each expression will only use these operators: {{operators}}.
```

crossover_user.txt.

```
Suggestion 1: {{assump1}}
Suggestion 2: {{assump2}}
Suggestion 3: {{assump3}}
Reference Expression 1: {{expr1}}
Reference Expression 2: {{expr2}}

Propose {{N}} expressions that would be appropriate given the suggestions and references.
Provide short commentary for each of your decisions. End with a JSON list that enumerates the
proposed expressions following this format:
```json
["expr1",
 "expr2",
 ...
 "expr{{N}}"
]
```
```

extract_idea_system.txt.

```
You are a helpful assistant that hypothesizes about the underlying assumptions that generated
a list of good and bad mathematical expressions in detailed ways. My ultimate goal is to
discover what assumptions generated the observed good mathematical expressions and excludes
the bad mathematical expressions. Focus more on the good expressions and their mathematical
structure. Note that capital C represents an arbitrary constant.
```

extract_idea_user.txt.

```
Good Expression 1: {{gexpr1}}
Good Expression 2: {{gexpr2}}
Good Expression 3: {{gexpr3}}
Good Expression 4: {{gexpr4}}
Good Expression 5: {{gexpr5}}

Bad Expression 1: {{bexpr1}}
Bad Expression 2: {{bexpr2}}
Bad Expression 3: {{bexpr3}}
Bad Expression 4: {{bexpr4}}
Bad Expression 5: {{bexpr5}}

Propose {{N}} hypotheses that would be appropriate given the expressions. Provide short
commentary for each of your decisions. Do not talk about topics related to the simplicity or
complexity of the expressions. I want ideas that are unique and interesting enough to amaze
the world's best mathematicians. End with a JSON list that enumerates the proposed hypotheses
following this format:
```json
["hyp1",
 "hyp2",
 ...
 "hyp{{N}}"
]
```
```

prompt_evol_system.txt.

```
You are a helpful assistant that merges and refines ideas about a set of hidden mathematical
expression in new, interesting, and diverse ways. My ultimate goal is to discover the
underlying properties of these hidden expressions. The resulting ideas should be a nontrivial
conclusion given the previous ideas.
```

prompt_evol_user.txt.

```
Idea 1: {{idea1}}
Idea 2: {{idea2}}
Idea 3: {{idea3}}
Idea 4: {{idea4}}
Idea 5: {{idea5}}

Propose {{N}} hypotheses that would be appropriate given the ideas. Provide short commentary
for each of your decisions. Do not talk about topics related to the simplicity or complexity
of the expressions. I want ideas that are unique and interesting enough to amaze the world's
best mathematicians. End with a JSON list that enumerates the proposed hypotheses following
this format:
```json
["hyp1",
 "hyp2",
 ...
 "hyp{{N}}"
]
```
```

E.4 RESTART PROMPTS

Message construction, conditional branches, and substitutions are specified in Appendix C.2.4.

Task descriptions.

```
Find the mathematical function skeleton that represents target value, given data on input
variable 1.

Find the mathematical function skeleton that represents target value, given data on input
variable 1, and input variable 2.
```

Neutral-context substitutions.

```
structures for scientific systems.
  -> structures from numerical data.
scientific meaning and relationships of inputs.
  -> mathematical relationships among the inputs.
scientifically meaningful
  -> mathematically meaningful
'Harmonic Oscillator'
  -> 'Trigonometric Function'
key characteristics in scientific context
  -> key mathematical characteristics
```

Candidate instruction.

```
You are a helpful assistant tasked with discovering mathematical function structures from
numerical data.                              Complete the 'equation' function below,
considering the mathematical relationships among the inputs.
```

Version docstring.

```
Improved version of `equation_v{i-1}`.
```

E2E feedback.

```
    Analysis for current equation(f):
        Given (X,y), Current equation(f) has MSE: {MSE:.6f},
        Definitions: current_pred = f_current(X) (the output of current equation).
        We fit a boosted equation g(X, current_pred) to approximate y.
        The learned boosted equation candidate is: {residual_eq},
        with MSE: {residual_mse:.6f}
```

Next-version instruction with boosted feedback.

```
    Consider previous versions of the equation with the corresponding boosted function and
    concepts(if any) for the problem,
        and create a new version of the equation that is more accurate.
         You can also combine terms from both the equation and boosted function from previous
         versions
```

Next-version instruction without boosted feedback.

```
    Consider previous versions of the equation and concepts(if any) for the problem,
        and create a new version of the equation that is more accurate.
         You can also combine terms from both the equation and function from previous versions
        while keeping the equation mathematically meaningful by retaining proper concepts.
```

Library-concept entries.

```
    Top scoring concepts for the problem:
        - {concept_name}, fit: {fit_score:.0f}/100
            definition: {definition}
            functions: {comma_separated_functions}
```

Concept-improvement message.

```
Analyze the improvement between the baseline equation and the new equation, then propose ONE
NEW mathematical concept that best explains the improvement.

BASELINE EQUATION (MSE: {baseline_mse:.6f}):
{baseline_code}

NEW EQUATION (MSE: {current_mse:.6f}):
{code}

IMPROVEMENT METRICS:
- Relative improvement: {improvement_ratio:.4f} ({improvement_ratio * 100:.2f}%)
- Absolute improvement: {absolute_improvement:.6f}
```

```
Based on this improvement, identify ONE key mathematical concept that likely contributed to
the better performance. Focus on:
1. New mathematical operations or functions introduced
2. Structural changes (e.g., polynomial degree, interaction terms)
3. Mathematical transformations or patterns

Provide:
- name: A concise mathematical concept name (e.g., 'Logarithmic Transform', 'Polynomial
Interaction')
- definition: Brief explanation of the concept and why it improves performance
- functions: List of specific functions/operators that implement this concept

Do not output other text, explanations, comments, or natural language.
Return STRICTLY ONE JSON object following the schema below.

JSON Schema: {schema_json}
```

Concept-output schema.

```
{
  "type": "object",
  "properties": {
    "name": {
      "type": "string",
      "description": "A concise, descriptive name for the mathematical concept (e.g., 'Power
      Law', 'Exponential Decay', 'Trigonometric Function')"
    },
    "definition": {
      "type": "string",
      "description": "A clear mathematical definition explaining what this concept represents
      and its key mathematical characteristics"
    },
    "functions": {
      "type": "string",
      "description": "The mathematical functions, operations, or transformations that
      characterize this concept, or relevant Python code implementations"
    }
  },
  "required": [
    "name",
    "definition",
    "functions"
  ]
}
```