Supplementary Material to "Univariate Skeleton Prediction in Multivariate Systems Using Transformers"

Anonymous Authors

No Institute Given

Abstract. In this supplementary file, we first explain the differences between our Multi-Set Transformer and the Set Transformer. Then, we provide details about the mathematical expression generation and the data generation processes proposed in this work. Finally, we report the experimental results obtained when comparing the symbolic skeletons generated by our Multi-Set Transformer and those obtained by PySR, TaylorGP, NeSymReS, and E2E.

A Set Transformer vs. Multi-Set Transformer

Here, we provide background information about the Set Transformer, the foundational basis for the Multi-Set Transformer. Furthermore, we enumerate the differences between the architectures and application domains of both models.

The Transformer [11] was designed to handle sequences with fixed orderings, where each element's position is crucial to understanding the data's meaning. In contrast, sets are collections of elements without any inherent order, and their permutations do not alter the underlying semantics. This inherent permutation invariance poses a significant obstacle for traditional sequence models when processing sets [13,4]. In order to address the limitations of conventional approaches, [9] presented an attention-based neural network module called Set Transformer that is based on the transformer model. This method introduces modifications to the transformer architecture, enabling it to handle sets without assuming a fixed ordering of elements.

A model designed for set-input problems must meet two essential criteria to effectively handle sets: First, it should be capable of processing input sets of varying sizes; second, it should exhibit permutation invariance. The latter means the output of the function represented by the model remains the same regardless of the order in which the elements of the input set are presented. More formally, Zaheer *et al.* [13] described this type of function as permutation equivariant:

Definition A.1. *Consider an input set* $\mathbf{x} = \{x_1, \ldots, x_n\}$*, where* $x_i \in \mathscr{X}$ *(i.e., the input domain is the power set* $\mathcal{X} = 2^{\mathscr{X}}$*). A function* $u : 2^{\mathscr{X}} \to \mathcal{Y}$ *acting on sets must be permutation invariant to the order of objects in the set, i.e. for any permutation* π*:*

$$u(\{x_1, \ldots, x_n\}) = u(\{x_{\pi(1)}, \ldots, x_{\pi(n)}\}),$$

where $\pi \in S_n$ *and* S_n *represents the set of all permutations of indices* $\{1, \ldots, n\}$*.*

The Set Transformer consists of two main parts: an encoder ϕ and a decoder ψ. The process starts by encoding the set elements in an order-agnostic manner. As such, the encoding for each element should be the same regardless of its position in the set. Then, ψ aggregates the encoded features and produces the desired output. Let us consider an input set $\mathbf{S} = \{\mathbf{s}_1, \ldots, \mathbf{s}_n\}$, where each element is d_{in}-dimensional ($\mathbf{S} \in \mathbb{R}^{n \times d_{in}}$). Therefore, the output T produced by the Set Transformer, whose computed function is denoted as g, can be expressed as:

$$T = g(\mathbf{S}) = \psi\left(\phi\left(\{\mathbf{s}_1, \ldots, \mathbf{s}_n\}\right)\right). \tag{1}$$

Typically, the order-agnostic property is achieved by making ϕ to act on each element of a set independently (i.e., $\psi\left(\{\phi(\mathbf{s}_1), \ldots, \phi(\mathbf{s}_n)\}\right)$) [13]. However, the Set Transformer uses self-attention mechanisms to encode the entire input set simultaneously to recognize interactions among the set instances.

For example, Lee *et al.* [9] used self-attention blocks (SABs), which, given an input set A, perform self-attention between the elements of the set and produce a set of equal size. This operation computes pairwise interactions among the elements of A; therefore, a stack of multiple SAB operations would encode higher-order interactions. One drawback of the use of SABs is that the attention mechanism is performed between two identical sets with n elements, which leads to a quadratic time complexity $\mathcal{O}(n)^2$. In order to alleviate this issue, Zaheer *et al.* [13] introduced the induced set attention block (ISAB). This could be interpreted as if the input set A was projected into a lower dimensional space and then reconstructed to produce outputs with the desired dimensionality. Since the attention mechanism involved in the ISAB operation is computed between a set of size m and a set of size n, its associated time complexity is $\mathcal{O}(mn)$. Furthermore, Lee *et al.* [9] proved that the SAB and ISAB blocks are universal approximators of permutation invariant functions.

It is worth noting that the Set Transformer was designed to produce fixed-size outputs. This setting can be used for applications such as population statistic estimation (e.g., retrieving a unique value that presents the median of an input set), unique character counting (i.e., obtaining the number of unique characters in a set of images), or k-amortized clustering where the objective is to produce k pairs of output parameters of a mixture of Gaussians (i.e., mean and covariance).

The most evident limitation of the Set Transformer, in the context of the problem discussed in this paper, resides in its encoder structure, which is specifically designed to process a single input set, as shown in Eq. 1. This is important because the input of the MSSP problem is defined as a collection of N_S input sets ($N_S > 1$). Hence, the encoder of our Multi-set Transformer is designed to process multiple input sets simultaneously. The Multi-Set Transformer architecture is depicted in Fig. A.1. Note that all input sets shown in this example were generated using the equation $y = \frac{1}{x\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{r}{x})^2}$ and a different r value was used for each set $\mathbf{D}^{(s)}$. Its architecture differs from the one used in the SSP method proposed by Biggio *et al.* [2], whose encoder only processes single input sets.

Note that the Set Transformer's output is k-dimensional; that is, its size is fixed depending on the problem. Conversely, the objective of the MSSP problem

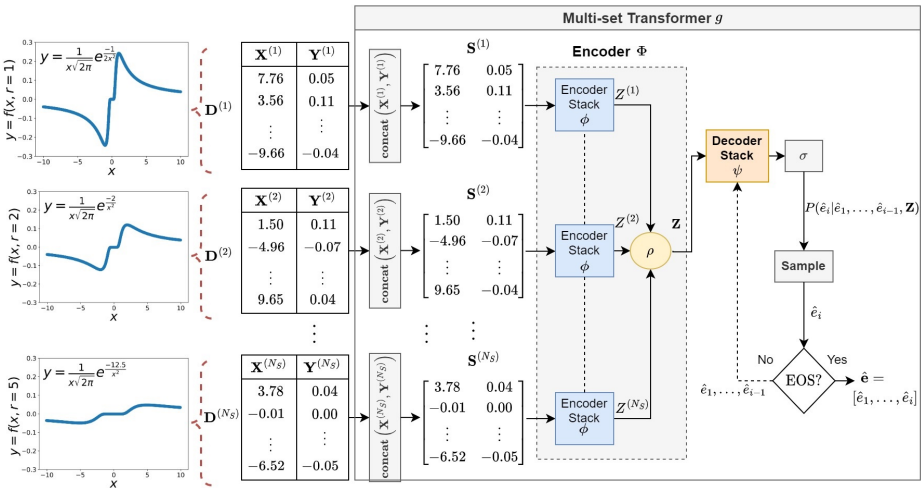

Fig. A.1: An example of an MSSP problem using the Multi-set Transformer.

is to generate a symbolic skeleton string whose length is not known *a priori* and depends on each input collection. Therefore, the decoder of the Multi-set Transformer is designed as a conditional-generative structure as it generates output sequences (i.e., the skeleton string) based on the encoded context.

B Data Generation

In this section, we explain how to generate the synthetic equations and corresponding data used to train our Multi-set Transformer. In addition, we explain the methodology we used to avoid generating functions that lead to undefined values during the training process.

B.1 Univariate Symbolic Skeleton Dataset Generation

Table B.1 provides the vocabulary used in this work. Our method differs from the generation method proposed by Biggio *et al.* [2], which, in turn, was based on that by Lample *et al.* [8]. Their method consists of constructing an expression tree with a specified number of total operators (i.e., unary and binary operators) by selecting operators and inserting them into random available places of the tree structure iteratively. The operators are chosen based on a pre-defined set of probability weights. In practice, we noticed that this approach leads to the generation of several expressions that contain binary operators exclusively and, thus, can be simplified into simple expressions such as $c_1\, x$ and $c_1\, x^2 + c_2$.

As an alternative, we use a generation method that builds the expression tree recursively in a preorder fashion (i.e., the root node is created before generating its left and right subtrees recursively). At each step, the algorithm decides

Table B.1: Vocabulary used to pre-train the Multi-set Transformer.

Token	Meaning	Token	Meaning
SOS	Start of sentence	sin	Sine
EOS	End of sentence	sinh	Hyperbolic sine
c	Constant placeholder	sqrt	Square root
x	Variable	tan	Tangent
abs	Absolute value	tanh	Hyperbolic tangent
acos	Arc cosine	-3	Integer number
add	Sum	-2	Integer number
asin	Arc sine	-1	Integer number
atan	Arc tangent	0	Integer number
cos	Cosine	1	Integer number
cosh	Hyperbolic cosine	2	Integer number
div	Division	3	Integer number
exp	Exponential	4	Integer number
log	Logarithmic	5	Integer number
mul	Multiplication	E	Euler's number
pow	Power	—	—

whether to add a binary operator, a unary operator, or a leaf node. Instead of assigning specific probability weights to individual operators, we assign weights to the types of nodes to be generated, namely unary operators, binary operators, or leaf nodes, with weights of 2, 1, and 1, respectively. In addition, rather than specifying the exact number of operators to include, we set a maximum limit on the total number of operators allowed in the expression. This adds flexibility in cases where including additional operators may incur violations of syntaxis or pre-defined conditions.

By generating the tree in a preorder manner, we can enforce certain conditions and constraints effectively. In particular, we can impose restrictions such as setting a maximum limit on the nesting depth of unary operators within each other. In addition, we can forbid certain combinations of operators. For example, we avoid embedding the operator `log` within the operator `exp`, or vice versa, since such composition could lead to direct simplification (e.g., $\texttt{log}\,(\texttt{exp}(x)) = x$). We can also avoid combinations of operators that would generate extremely large values (e.g., $\texttt{exp}\,(\texttt{exp}(x))$ and $\texttt{sinh}\,(\texttt{sinh}(x))$). Table B.2 shows the forbidden operators we considered for some specific parent operators.

Each generated tree is traversed to derive a mathematical expression in prefix notation. The expression is then transformed from prefix to infix notation, which is simplified using the SymPy library[1]. The simplified expression is transformed again into an expression tree. Then, each non-numerical node of the tree is multiplied by a unique placeholder and then added to another unique placeholder. Some exceptions apply; for instance, the arguments of the `exp`, `sinh`, `cosh`, and `tanh` operators are not affected by additive placeholders. This is be-

[1] `https://www.sympy.org/`

Table B.2: Forbidden combinations of operators

Parent Node	Forbidden Operators
abs	sqrt, pow2, pow4
exp, tan, ln	exp, sinh, cosh, tanh, tan, ln, pow3, pow4, pow5
sinh, cosh, tanh	exp, sinh, cosh, tanh, tan, ln, pow2, pow3, pow4, pow5
pow2, pow3, pow4, pow5	pow2, pow3, pow4, pow5, exp, sinh, cosh, tanh
sin, cos, tan	sin, cos, tan
asin, acos, atan	asin, acos, atan

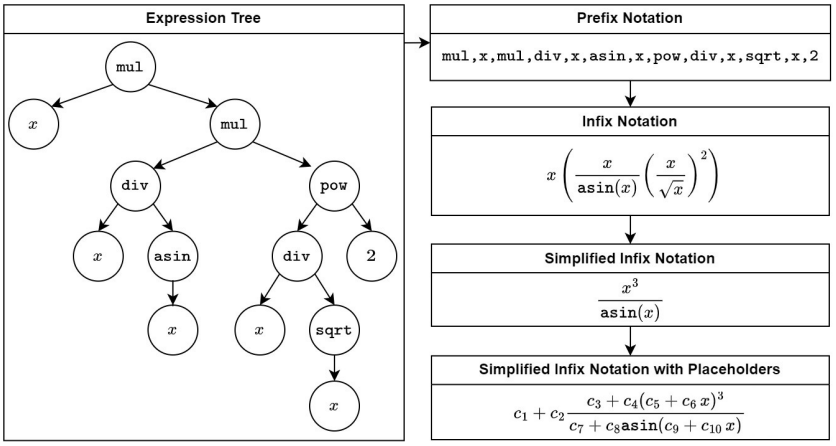

Fig. B.1: Example of a randomly generated expression.

cause adding a constant to their arguments would lead to direct simplification (e.g., $c_1 e^{c_2\,x+c_3} = (c_1 e^{c_2}) e^{c_3\,x}$). Fig. B.1 illustrates the random expression generation process. Note that previous approaches also proposed to include unique placeholders in the generated expressions; however, they included only one placeholder at a time and in some of the nodes. In contrast, our approach allows for the generation of more general expressions.

B.2 Avoiding Invalid Operations

In this section, we present a detailed description of the function $\texttt{avoidNaNs}(\mathbf{x}, f)$, initially introduced in Algorithm 2 of the main paper. This function serves the purpose of modifying certain coefficients within the function f or generating supplementary support values within the vector $\mathbf{x}$ to avoid numerical inconsistencies. We classify the operators that may generate undefined values as follows:

– Single-bounded operators: These operators have bounded numerical arguments due to the mathematical constraints imposed on their domains. We consider the following single-bounded operators:

- Logarithm: The `log` operator is bounded on its left side as it cannot receive an input lower than or equal to 0:

$$\text{Domain}\,(\texttt{log}(x)) = \{x \in \mathbb{R} \mid x > 0\}.$$

- Square root: The `sqrt` operator is also bounded on its left side to avoid generating complex numbers:

$$\text{Domain}\,(\texttt{sqrt}(x)) = \{x \in \mathbb{R} \mid x \geq 0\}.$$

- Exponential: We decided to bound the `exp` operator on its right side to avoid generating extremely large values. This also applies to the `sinh`, `cosh`, and `tanh` operators:

$$\text{Domain}\,(\texttt{exp}(x)) = \{x \in \mathbb{R} \mid x \leq 7\}.$$

– Double-bounded operators: Unlike the single-bounded operators, the numerical arguments of these operators are bounded on their left and right sides. We consider the following double-bounded operators:
  - Arcsine: The `asin` operator takes a value between -1 and 1 as its input and returns the angle whose sine is equal to that value:

$$\text{Domain}\,(\texttt{asin}(x)) = \{x \in \mathbb{R} \mid -1 \leq x \leq 1\}.$$

  - Arccosine: Like arcsine, the `acos` also takes a value between -1 and 1 as its input operator:

$$\text{Domain}\,(\texttt{acos}(x)) = \{x \in \mathbb{R} \mid -1 \leq x \leq 1\}.$$

– Operators with singularities: Operators like tangent or division can exhibit singularities at specific input values. For instance, the tangent function becomes undefined when its input equals an odd multiple of $\pi/2$, resulting in an asymptotic behavior where the function approaches infinity.

The method proposed to address the NaN ("not a number") values arising from the aforementioned functions, which we refer to as "special functions", is shown in Algorithm B.1. Here, f.`args` returns a vector containing the numerical arguments of function f. If f is considered as an expression tree, f.`func` returns the name of the operator located at the top of the tree. The algorithm analyzes each argument of f separately (Line 3). A given argument of f, denoted as arg, can be considered as a sub-expression tree. Hence, the function `containSpecialF`(arg) (Line 4) traverses the sub-expression tree and returns a true value if any special function is found in it.

If the current sub-expression tree contains a special function, we check if the top operator of the subtree is special using function `isSpecialF`(arg.`func`) (Line 5). If not, we move down to a deeper level of the subtree using recursion (Line 6). Otherwise, the arguments of the sub-expression arg may need to be modified to avoid undefined values. Before doing so, in Line 8, we verify if

Algorithm B.1 Avoiding Invalid Operations

```
1: function AVOIDNANS(x, f, Xsing = [])
2:     args, newArgs ← f.args, []
3:     for each arg ∈ args do
4:         if containSpecialF(arg) then
5:             if !isSpecialF(arg.func) then                    ▷ If the top operator is special
6:                 x, arg, Xsing ← avoidNaNs(x, arg, Xsing)
7:             else
8:                 if containSpecialF(arg.args[0]) then         ▷ If there's a special function inside
9:                     x, arg, Xsing ← avoidNaNs(x, arg, Xsing)
10:                innArg ← arg.args[0]
11:                vals ← arg(x)
12:                if containNaNs(vals) then
13:                    innVals ← innArg(x)
14:                    if isSingleBounded(arg) then
15:                        innArg ← modifySBounded(arg, vals, innArg, innVals)
16:                    else if isDoubleBounded(arg) then
17:                        innArg ← modifyDBounded(arg, innArg, innVals)
18:                    else                                     ▷ Operations with singularities
19:                        x, sing ← avoidSingularities(length(x), arg, Xsing)
20:                arg.args[0] ← innArg                         ▷ Update function
21:            newArgs.append(arg)
22:        f.args ← newArgs
23:        return x, arg, sing
```

there is another special function contained inside the current subtree, in which case we explore a deeper level of the subtree using recursion. Note that we apply the `containSpecialF`(arg.`args`[0]) function given that arg is guaranteed to represent a special function; i.e., it is a unary function with a single argument.

The function arg is evaluated on the input values $\mathbf{x}$ and the obtained values are stored in the vector $vals$ (Line 11). The function `containNaNs`($vals$) returns a true value in the event that one or more undefined values are found within the vector $vals$. If an undefined value is found, a modification of the inner argument, $innArg$, or the input vector $\mathbf{x}$ is needed. To do so, we first evaluate the function represented by $innArg$ on the input values $\mathbf{x}$ and store the outcomes in the variable $innVals$ (Line 13). In the case that the function arg is single-bounded, the domain of function $innArg$ is modified accordingly using `modifySBounded`($arg, vals, innArg, innVals$) (Line 15) by adding a horizontal offset. Likewise, if arg is double-bounded, the domain of $innArg$ is modified accordingly using `modifyDBounded`($arg, innArg, innVals$) (Line 17) by re-scaling $innArg$. In the case that arg represents a function with singularities, a new input vector $\mathbf{x}$ with resampled values is obtained using the function `avoidSingularities`($\mathbf{x}, arg, Xsing$) (Line 19), where $Xsing$ is a variable that will store all positions at which the function produces undefined values. Fig. B.2 depicts an example of how data are generated using the underlying function $f(x) = \frac{-3.12x}{\texttt{sin}(1.45x)} - 2.2$. The figure at the bottom shows in detail that the `avoidSingularities` function generates valid intervals of values for $\mathbf{x}$ by avoiding getting too close to the singular points (red dotted line). Furthermore, the inner argument of function arg is then replaced with the modified function $innArg$ (Line 21). Finally, the arguments of the original function f are replaced with the modified arguments stored in the list $newArgs$ (Line 22).

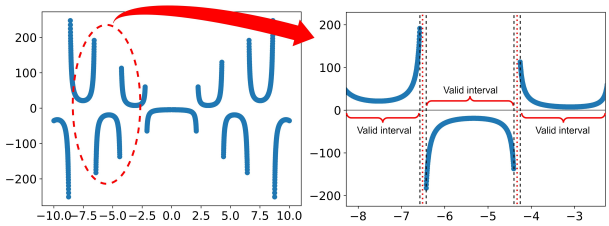

Fig. B.2: Generation data for $f(x) = \frac{-3.12x}{\sin(1.45x)} - 2.2$. (Left) Generated data on the entire domain. (Right) Detailed view of how singularities are avoided.

Table B.3: Equations used for experiments

Eq.	Underlying equation	Reference	Domain range
E1	$(3.0375x_1x_2 + 5.5\sin(9/4(x_1 - 2/3)(x_2 - 2/3)))/5$	[6]	$[-5, 5]^2$
E2	$5.5 + (1 - x_1/4)^2 + \sqrt{x_2 + 10}\sin(x_3/5)$	—	$[-10, 10]^2$
E3	$(1.5e^{1.5x_1} + 5\cos(3x_2))/10$	[6]	$[-5, 5]^2$
E4	$(1 - x_1)^2 + (1 - x_3)^2 + 100(x_2 - x_1^2)^2 + 100(x_4 - x_3^2)^2$	Rosenbrock-4D	$[-5, 5]^4$
E5	$\sin(x_1 + x_2x_3) + \exp(1.2x_4)$	—	$x_1 \in [-5, 5]$, $x_2 \in [-3, 3]$, $x_3 \in [-3, 3]$, $x_4 \in [-3, 3]$
E6	$\tanh(x_1/2) + \lvert x_2\rvert\cos(x_3^2/5)$	—	$[-10, 10]^3$
E7	$(1 - x_2^2)/(\sin(2\pi\, x_1) + 1.5)$	[12]	$[-5, 5]^2$
E8	$(5x_1)^4/((5x_1)^4 + 1) + (5x_2)^4/((5x_2)^4 + 1)$	[10]	$[-5, 5]^2$
E9	$\log(2x_2 + 1) - \log(4x_1^2 + 1)$	[10]	$[0, 5]^2$
E10	$\sin(x_1\, e^{x_2})$	[1]	$x_1 \in [-2, 2], x_2 \in [-4, 4]$
E11	$x_1\, \log(x_2^4)$	[1]	$[-5, 5]^2$
E12	$1 + x_1\, \sin(1/x_2)$	[1]	$[-10, 10]^2$
E13	$\sqrt{x_1}\, \log(x_2^2)$	[1]	$x_1 \in [0, 20], x_2 \in [-5, 5]$

C Comparison of Predicted Symbolic Skeletons

In this section, we report the symbolic skeletons predicted by two GP-based SR methods, PySR [3] and TaylorGP [5], and three neural SR methods, NeSymRes [2], E2E [7], and our Multi-Set Transformer. For ease of reference, we reproduce the list of SR problems used for our experiments in Table B.3.

Tables C.1 and C.2 report the skeletons obtained by all compared methods. It is important to note that these experiments are designed to validate the functional form obtained between each system's variable and the system's response. That is, instead of evaluating the expressions generated by the compared methods directly, we first extract their skeletons with respect to each system's variable. For example, NeSymRes produces the estimated function $\hat{f}(x_1, x_2) = 0.592\, x_1\, x_2 + \cos(0.014\,(-0.995\, x_1 + x_2 - 0.08)^2)$ for problem E1. From this, the skeletons with respect to variables x_1 and x_2 are $\hat{e}(x_1) = c_1\, x_1 + \cos(c_2(c_3 + c_4\, x1)^2)$ and $\hat{e}(x_2) = c_1\, x_2 + \cos(c_2(c_3 + x_2)^2)$, respectively.

Table C.1: Comparison of skeleton prediction results (E1–E9)

Prob.	Met. / Var.	PySR	TaylorGP	NeSymReS	E2E	MST	Target $\mathbf{e}(x)$
E1	x_1	$c_1\,x_1$	$c_1\,x_1$	$c_1\,x_1 + \cos(c_2(c_3 + c_4\,x_1)^2)$	$c_1 + c_2\,x_1 + c_3\sin((c_4 + c_5\,x_1)^2)$	$c_1 + c_2\,x_1 + c_3\sin(c_4 + c_5\,x1)$	$c_1\,x_1 + c_2\,\sin(c_3(c_4 + x_1))$
	x_2	$c_1\,x_2$	$c_1\,x_2$	$c_1\,x_2 + \cos(c_2(c_3 + x_2)^2)$	$c_1 + c_2(c_3 + c_4\,x_2)$	$c_1 + c_2\,x_2 + c_3\sin(c_4 + c_5\,x_2)$	$c_1\,x_2 + c_2\,\sin(c_3(c_4 + x_2))$
E2	x_1	$c_1 + \lvert c_2 + \lvert c_3 + x_1\rvert\rvert$	$c_1 + c_2\,x_1$	$c_1 + c_2\,x_1$	$c_1 + c_2\,x_1 + c_3(c_4 + c_5\,x_1)^2$	$c_1 + c_2(c_3 + c_4\,x_1)^2$	$c_1 + (c_2 + c_3\,x_1)^2$
	x_2	c_1	c_1	$c_1 + e^{e^{c_2\,x_2}}$	$c_1 + c_2\,(c_3 + c_4\,x_2)$	$c_1\sqrt{c_2\,x_2 + c_3} + c_4$	$c_1\sqrt{x_2 + c_2} + c_3$
	x_3	$c_1 + c_2\,x_3$	$c_1 + c_2\,x_3$	$c_1 + c_2\,x_3$	$c_1 + c_2\,x_3 + c_3(c_4 + c_5\cos(c_6 + c_7\,x_3))$	$c_1 + c_2\sin(c_3\,x_3 + c_4)$	$c_1 + c_2\sin(c_3\,x_3)$
E3	x_1	$c_1\,e^{x_1}\,\lvert\sinh(c_2\,x_1)\rvert$	$c_1\,e^{c_2\,x_1}$	$c_1\,e^{c_2\,x_1}$	$c_1 + c_2 e^{c_3\,x_1}$	$c_1 + c_2 e^{c_3\,x_1}$	$c_1 + c_2\,e^{c_3\,x_1}$
	x_2	c_1	c_1	$c_1\,\cos(c_2\,x_2)$	$c_1 + c_2\,\cos(c_3 + c_4\,x_2)$	$c_1 + c_2\,\cos(c_3 + c_4\,x_2)$	$c_1 + c_2\,\cos(c_3\,x_2)$
E4	x_1	$c_1 + c_2\,x_1^2$	$c_1\,x_1^2$	—	$c_1 + c_2\lvert c_3 + c_4\,x_1 + c_5\,x_1^2 + c_6\,x_1^3 + c_7\,x_1^4\rvert$	$c_1(c_2\,x1 + c_3)^4 + c_4$	$c_1 + c_2\,x_1 + c_3\,x_1^2 + c_4\,x_1^4$
	x_2	$c_1 + c_2\,x_2$	c_1	—	$c_1 + c_2\lvert c_3 + c_4\,x_2 + c_5\,x_2^2\rvert$	$c_1(c_2\,x2 + c_3)^2 + c_4$	$c_1 + c_1\,x_2 + c_3\,x_2^2$
	x_3	$c_1 + c_2\,x_3^2$	c_1	—	$c_1 + c_2\lvert c_3 + c_4\,x_3 + c_5\,x_3^2 + c_6\,x_3^3 + c_7\,x_3^4\rvert$	$c_1(c_2\,x3 + c_3)^4 + c_4$	$c_1 + c_2\,x_3 + c_3\,x_4^2 + c_4\,x_3^4$
	x_4	$c_1 + c_2\,x_4$	c_1	—	$c_1 + c_2\lvert c_3 + c_4\,x_4 + c_5\,x_4^2\rvert$	$c_1(c_2\,x4 + c_3)^2 + c_4$	$c_1 + c_2\,x_4 + c_3\,x_4^2$
E5	x_1	c_1	c_1	—	$c_1 + c_2\cos(c_3 + c_4\,x_1)$	$c_1\sin(c_2\,x1 + c_3) + c_4$	$c_1 + \sin(c_2 + c_3\,x_1)$
	x_2	c_1	c_1	—	$c_1 + c_2\,x_2$	$c_1\sin(c_2\,x2 + c_3) + c_4$	$c_1 + \sin(c_2 + c_3\,x_2)$
	x_3	c_1	c_1	—	$c_1 + c_2\,x_3 + c_3\,x_3^2 + c_4\,x_3^3$	$c_1\sin(c_2\,x3 + c_3) + c_4$	$c_1 + \sin(c_2 + c_3\,x_3)$
	x_4	$e^{c_1\,x_4}$	$c_1\,e^{x_4}\,e^{\sin(c_2\,x_4)}$	—	$c_1 + c_2\,e^{c_3\,x_4}$	$c_1 e^{c_2\,x4} + c_3$	$c_1 + e^{c_2\,x_4}$
E6	x_1	c_1	c_1	$c_1 + c_2\,x_1$	$c_1 + c_2\,\mathrm{atan}(c_3 + c_4\,x_1)$	$c_1 + c_2\tanh(c_3\,x_1)$	$c_1 + \tanh(c_2\,x_1)$
	x_2	c_1	c_1	$c_1 + x_2\,\sin(c_2/(c_3 + x_2))$	$c_1 + c_2\,x_2 + c_3\,x_2\,\cos(c_4 + c_5\,x_2)$	$c_1 + c_2\,\lvert x_2\rvert$	$c_1 + c_2\,\lvert x_2\rvert$
	x_3	$\tanh(\exp(x_3))$	$c_1\,\sin(c_2\,x_3^2)/(c_3\sqrt{x_3} + \sin(\sqrt{x_3}))$	c_1	c_1	$c_1\,\cos(c_2\,(c_3 + x_3)^2) + c_4$	$c_1 + c_2\,(\cos(c_3\,x_3^2))$
E7	x_1	$c_1 + x_1^2$	c_1	$c_1 + c_2\,x_1$	$c_1 + c_2\,\sin(c_3 + c_4\,x_1)^2 + c_5\,\sin(c_6 + c_7\,x_1)$	$c_1 + c_2/(c_3 + \sin(c_4\,x_1 + c_5))$	$c_1/(c_2 + \sin(c_3\,x_1))$
	x_2	$c_1/\sinh(\sinh(\tanh(e^{\sinh(\sin(c_2\,x_2))})))$	$c_1\,x_2^2 + \sqrt{\lvert x_2\rvert}$	$(c_1 + x_2^2)/(c_2 + \cos(c_3\,(c_4 + c_5\,x_2)^3))$	$c_1\,(c_2 + c_3\,x_2)^2$	$c_1 + c_2\,x_2^2$	$c_1 + c_2\,x_2^2$
E8	x_1	$c_1 + \tanh(c_2 + \cosh(x_1))$	c_1	$e^{c_1\,\cos(c_2/x_1)}$	$c_1 + c_2\,ec_3\,\lvert c_4 + c_5\,x_1\rvert$	$c_1 + c_2/(c_3\,x_1^4 + c_4)$	$c_1 + c_2\,x_1^4/(c_3 + c_4\,x_1^4)$
	x_2	$c_1 + \cos(\tan(\tanh(c_2/x_2^2)))$	c_1	$c_1 + \cos(1/x_2)$	$c_1 + c_2\,e^{c_3\,\lvert c_4 + x_2\rvert}$	$c_1 + c_2/(c_3\,x_1^4 + c_4)$	$c_1 + c_2\,x_2^4/(c_3 + c_4\,x_2^4)$
E9	x_1	$\log(c_1/(c_2 + c_3\,x_1^2))$	$c_1 + \log(c_2/\lvert x_1\rvert)$	$\log(c/\lvert x_1\rvert)$	$c_1 + c_2\,\log(c_3 + c_4\,x_1 + c_5\,x_1^2)$	$c_1 + \log(c_2\,x_1^2 + c_3)$	$c_1 + c_2\,\log(c_3 + c_4\,x_1^2)$
	x_2	$\log(c_1 + c_2\,x_2)$	$c_1\exp(c_2\,x_2)$	$\log(c\,\lvert x_2\rvert)$	$c_1 + c_2\,\log(c_3 + c_4/(c_5 + c_6\,x_2 + c_7\,x_2^2))$	$c_1 + \log(c_2 + c_3\,x_2)$	$c_1 + \log(c_2 + c_3\,x_2)$

Table C.2: Comparison of skeleton prediction results (E10–E13)

Prob.	Var.	PySR	TaylorGP	NeSymReS	E2E	MST	Target $\mathbf{e}(x)$
E10	x_1	$\sin(c_1\,x_1)$	$c_1\,x_1$	$\sin(c_1\,x_1)$	$c_1 + c_2\,\sin(c_3 + c_4\,x_1)$	$c_1 + c_2\,\sin(c_3\,x_1)$	$\sin(c_1\,x_1)$
	x_2	$\sin(c_1\,e^{x_2})$	$c_1\,e^{c_2\sqrt{\lvert x_2\rvert}}$	$\sin(c_1\,e^{x_2})$	$c_1 + c_2\,\sin(c_3 + c_4\,e^{c_5\,x_2})$	$c_1 + \sin(c_2\,e^{c_3\,x_2})$	$\sin(c_1\,e^{x_2})$
E11	x_1	$c_1\,x_1$	$c_1\,x_1$	$c_1\,x_1$	$c_1\,x_1$	$c_1 + c_2\,x_1$	$c_1\,x_1$
	x_2	$c_1\,\log(x_2^4)$	$c_1\,\log(\lvert x_2\rvert)$	$c_1\,\log(x_2^4)$	$c_1 + c_2\,\log(c_3 + c_4\,\lvert c_5 + c_6\,x_2\rvert)$	$c_1 + c_2\,\log(c_3\,x_2^2)$	$c_1\,\log(x_2^4)$
E12	x_1	$c_1 + \sin(x_1)\,\lvert x_1\rvert$	$c_1 + c_2\,x_1$	$c_1 + c_2\,x_1$	$c_1 + c_2\,x_1$	$c_1 + c_2\,x_1$	$c_1 + c_2\,x_1$
	x_2	$c_1 + c_2\,\sin(c_3/x_2)$	$c_1\,\sin(c_2/x_2) + \sqrt{x_2}\,\sin(c_3/x_2)$	$c_1\,\sin(1/x_2) + x_2\,\sin(1/x_2)$	$c_1\,\sin(c_2/(c_3 + c_4\,x_2))$	$c_1 + c_2\,\sin(c_3/x_2)$	$c_1 + c_2\,\sin(1/x_2)$
E13	x_1	$c_1\,\sqrt{x_1}$	$c_1 + c_2\,\sqrt{e^{\sqrt{x_1}}}$	$c_1 + c_2\,x_1$	$c_1 + c_2\,\log(c_3 + c_4\,x_1)$	$c_1 + c_2\,\sqrt{c_3 + x_1}$	$c_1\,\sqrt{x_1}$
	x_2	$c_1\,\log(x_2^2)$	$c_1 + \log(\lvert x_2\rvert) + c_2\,\log(\lvert x_2\rvert)$	$c_1\,\log(x_2^2)$	$c_1 + c_2/(c_3 + c_4\,\lvert c_5 + c_6\,x_2\rvert)$	$c_1 + c_2\,\log(c_3\,(c_4 + x_2)^2)$	$c_1\,\log(x_2^2)$

References

1. Bertschinger, A., Davis, Q.T., Bagrow, J., Bongard, J.: The metric is the message: Benchmarking challenges for neural symbolic regression. In: Machine Learning and Knowledge Discovery in Databases. pp. 161–177 (2023)
2. Biggio, L., Bendinelli, T., Neitz, A., Lucchi, A., Parascandolo, G.: Neural symbolic regression that scales. In: Proceedings of the 38th International Conference on Machine Learning. vol. 139, pp. 936–945 (2021)
3. Cranmer, M.: Interpretable machine learning for science with PySR and SymbolicRegression.jl. ArXiv **abs/2305.01582** (2023)
4. Edwards, H., Storkey, A.: Towards a neural statistician. In: International Conference on Learning Representations. Toulon, France (2017)
5. He, B., Lu, Q., Yang, Q., Luo, J., Wang, Z.: Taylor genetic programming for symbolic regression. In: Proceedings of the Genetic and Evolutionary Computation Conference. p. 946–954 (2022)
6. Jin, Y., Fu, W., Kang, J., Guo, J., Guo, J.: Bayesian symbolic regression. ArXiv **abs/1910.08892** (2020)
7. Kamienny, P.A., d'Ascoli, S., Lample, G., Charton, F.: End-to-end symbolic regression with transformers. In: Advances in Neural Information Processing Systems. vol. 35, pp. 10269–10281 (2022)
8. Lample, G., Charton, F.: Deep learning for symbolic mathematics. In: International Conference on Learning Representations (2020)
9. Lee, J., Lee, Y., Kim, J., Kosiorek, A., Choi, S., Teh, Y.W.: Set Transformer: A framework for attention-based permutation-invariant neural networks. In: Proceedings of the 36th International Conference on Machine Learning. vol. 97, pp. 3744–3753 (2019)
10. Trujillo, L., Muñoz, L., Galván-López, E., Silva, S.: neat genetic programming: Controlling bloat naturally. Information Sciences **333**, 21–43 (2016)
11. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A., Kaiser, L., Polosukhin, I.: Attention is all you need. In: Advances in Neural Information Processing Systems. vol. 30 (2017)
12. Werner, M., Junginger, A., Hennig, P., Martius, G.: Informed equation learning. ArXiv **abs/2105.06331** (2021)
13. Zaheer, M., Kottur, S., Ravanbakhsh, S., Poczos, B., Salakhutdinov, R.R., Smola, A.J.: Deep sets. In: Guyon, I., Luxburg, U.V., Bengio, S., Wallach, H., Fergus, R.,

Vishwanathan, S., Garnett, R. (eds.) Advances in Neural Information Processing Systems. vol. 30. Curran Associates, Inc. (2017)