David Svetoslavov Yordanov

Characterization of low dimensional embeddings for the generation of closed-form mathematical expressions.

Degree Final Project

Supervised by Dr Marta Sales Pardo
and Dr Roger Guimerà Manrique

Bachelor's degree in Mathematics and Physics Engineering

UNIVERSITAT ROVIRA i VIRGILI

Tarragona
2025

Abstract

This bachelor's degree project aims to present and discuss a number of quantitative experiments carried out with the objective of better understanding and evaluating the performance of a *Hierarchical Variational Autoencoder* (HVAE) in the context of *symbolic regression*. HVAE is a recently proposed machine learning model that encodes expression trees into a continuous latent space from which it can then decode new ones.

The experiments conducted in this project are split into two groups: latent space characterization and symbolic regression. The first group consists of random sampling, random walks and encode-decode reconstruction experiments. Their aim is to gain insight into the structure of the latent space and how encoded expressions are distributed within it. The symbolic regression experiments are carried out using simple error minimization algorithms, which will test the effectiveness of HVAE as a plausible expression generator.

Keywords: Symbolic Regression, Machine Learning, Autoencoders

Resumen

Este proyecto de trabajo de fin de grado tiene como objetivo presentar y discutir una serie de experimentos cuantitativos realizados con el propósito de comprender mejor y evaluar el rendimiento de un *Autoencoder Variacional Jerárquico* (HVAE) en el contexto de la *regresión simbólica*. HVAE es un modelo de aprendizaje automático propuesto recientemente que codifica árboles de expresiones en un espacio latente continuo desde el cual puede decodificar nuevas expresiones.

Los experimentos llevados a cabo en este proyecto se dividen en dos grupos: caracterización del espacio latente y regresión simbólica. El primer grupo consiste en muestreo aleatorio, caminatas aleatorias y experimentos de reconstrucción codificar-decodificar. Su objetivo es obtener información sobre la estructura del espacio latente y cómo se distribuyen en él las expresiones codificadas. Los experimentos de regresión simbólica se realizan utilizando algoritmos simples de minimización de error, los cuales ponen a prueba la efectividad del HVAE como generador de expresiones plausibles.

Palabras clave: Regresión simbólica, Aprendizaje automático, Autoencoders

Resum

Aquest projecte de treball de fi de grau té com a objectiu presentar i discutir una sèrie d'experiments quantitatius realitzats amb la finalitat de comprendre millor i avaluar el rendiment d'un *Autoencoder Variacional Jeràrquic* (HVAE) en el context de la *regressió simbòlica*. El HVAE és un model d'aprenentatge automàtic proposat recentment que codifica arbres d'expressions en un espai latent continu des del qual pot posteriorment decodificar-ne de nous.

Els experiments realitzats en aquest projecte es divideixen en dos grups: caracterització de l'espai latent i regressió simbòlica. El primer grup consisteix en mostreig aleatori, passejades aleatòries i experiments de reconstrucció codificació-decodificació. El seu objectiu és obtenir informació sobre l'estructura de l'espai latent i com s'hi distribueixen les expressions codificades. Els experiments de regressió simbòlica es porten a terme utilitzant algorismes simples de minimització d'error, que posaran

a prova l'efectivitat del HVAE com a generador d'expressions plausibles.

Paraules clau: Regressió simbòlica, Aprenentatge automàtic, Autoencoders

Acknowledgments

Special thanks are extended to Dr. Roger Guimerà Manrique and Dr. Marta Sales Pardo for their guidance and support throughout the course of this project. Their expertise in the field has been key to the development and completion of this work. Moreover, the intellectually stimulating and pleasant work environment they create has made this research experience both highly enriching and personally rewarding.

Words of appreciation go to Dr. Juan Alberto Rodriguez Velázquez for tutoring me during my years as a bachelor's student. His advice and motivation during my first years in university have been highly significant in the development of my academic career.

I am sincerely grateful to my family for their unconditional support, for granting me the opportunity of getting into higher education, and for providing the necessary resources needed for my studies.

Finally, thank you to Sebastian Meznar and his team for developing and providing the HVAE model, which was the main focus of this study.

Contents

1 Introduction 1

1.1 Objectives 2

2 Background 3

2.1 The symbolic regression problem 3
2.2 Symbolic regression's biggest challenges 4
2.3 Expression trees 6
2.4 Hierarchical Variational Autoencoder 8

3 Results 11

3.1 Latent space characterization experiments 11
3.1.1 Random *l-space* sampling 11
3.1.2 Random walks 14
3.1.3 Encode-decode reconstruction 21
3.2 Symbolic regression experiments 24
3.2.1 Restricted random walk 25
3.2.2 Simulated annealing 29

4 Conclusions 32

A Code Availability 34

1 Introduction

Symbolic regression (SR) is a machine learning task that aims to discover underlying mathematical relationships between dependent and independent variables from a given set of data [1]. In general, classical regression techniques look to optimize parameters of a model chosen *a priori*, while SR works without any prior intuition or assumption about the given data [2]. This means that relationships discovered by symbolic regression algorithms aren't affected by human bias that might be introduced when selecting a specific model to be optimized. Instead, SR uncovers knowledge purely from patterns in the data.

The way the relationships are represented is through functions that map independent variables to dependent ones. Said functions can be of many different types, but this study — just like most practical applications of SR — will focus on ones that take a vector of real numbers as input and generate a real-valued scalar as output. Mathematically, these are represented as closed-form expressions where the value of the dependent variable is obtained from a mathematical formula in terms of the independent variables. A simple example is $y = x^2 + 1$, where y is the dependent variable and x is the independent one. Symbolic regression algorithms aim to generate expressions similar to the one on the right-hand-side of the equality in the previous example that also fit the values of a given dataset. More formalism on this topic will be presented in section 2.1.

The research in symbolic regression has been developing since the last decades of the 20th century, when *genetic programming* (GP) began to be used among researchers to generate and find mathematical expressions that fit well to a given dataset. Algorithms were implemented using evolutionary operators like *mutation* and *crossover* on populations of expressions in order to explore solution spaces with greater efficiency. Advancements in machine learning and computational power during the 2000s and 2010s have allowed researchers to improve the performance of SR algorithms, and new approaches such as using neural networks or Bayesian methods [3] to evaluate the quality of proposed solutions have also been presented and implemented.

As one might expect, symbolic regression is not a simple problem to solve, especially when working with noisy data. In fact, it falls into the category of NP-hard[1] problems due to the infinite number of possible expressions that can be considered as a possible solution in any given case [4]. Constraints on the complexity of the expressions can be applied in order to reduce the solution space size for certain problems, but finding computationally efficient ways to explore general solution spaces is one of the biggest challenges in SR.

This study was started with the aim of improving the performance of the **Bayesian Machine Scientist** (BMS) model for symbolic regression [3]. Broadly speaking, BMS generates expressions by sampling candidates from an estimated probability distribution over the space of mathematical expressions.

A recently proposed machine learning model aims to improve the process of generating candidate expressions to be considered as possible solutions to an SR problem. The model is a **Hierarchical Variational Autoencoder** (HVAE) that introduces a novel way of encoding mathematical expressions into a continuous latent space from which new expressions can later be decoded [1] . HVAE is an open source model that may improve the efficiency with which the domain of possible solutions to an SR problem is explored.

[1] Problems for which it is not possible to always find an optimal solution in polynomial time.

1.1 Objectives

The motivation behind this research is that HVAE could prove more efficient than BMS in the generation of candidate expressions. If true, combining HVAE and BMS could result in a more effective symbolic regression model.

With this goal in mind, we can establish the following objectives for this project:

1. Learn how to implement HVAE and manipulate its parameters in order to adapt the model to the problem at hand.
2. Gain insight into the structure of the encoded latent spaces of HVAE.
3. Successfully implement a symbolic regression model using HVAE as an expression generator.
4. Evaluate the performance of HVAE as a possible expression generator.

It is important to state that this project **does not** aim to study the underlying theory and machine learning methods that HVAE uses to encode and decode expressions. Technical details on the design of HVAE can be found in [1]. This study will treat HVAE as a customizable, ready-for-use tool that has been presented to the symbolic regression community.

2 Background

This section aims to give an introduction to the fundamental concepts that are relevant for this study. We will start by providing a formal definition of the problem of symbolic regression, then we'll talk about why we study it, what are its main challenges and what methodologies are being used to solve it. Afterward, an introduction to the HVAE model will be given and we'll talk about its functionality, what makes it an interesting study topic and the performance benchmarks given by its creators.

2.1 The symbolic regression problem

We begin by giving a formal definition of the problem of symbolic regression for the case of real-numbered data. Let $\mathbf{x} = (x_1, x_2, \ldots, x_n) \in \mathbb{R}^n$ be a vector that contains the observed values for n different independent variables and $y \in \mathbb{R}$ be the observed value of the dependent variable. The pair $(\mathbf{x}, y)$ represents the correspondence between the variables, i.e. the value y corresponds to the independent values $\mathbf{x}$. The set $\mathcal{O} \equiv \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_k, y_k)\}$ is the collection of observed data, which can be visualized as k points sitting in $\mathbb{R}^{n+1}$ space.

A **symbolic regression** (SR) algorithm takes $\mathcal{O}$ as input and aims to output a function $f : \mathbb{R}^n \to \mathbb{R}$ such that $f(\mathbf{x}_i) = y_i,\ \forall(\mathbf{x}_i, y_i) \in \mathcal{O}$. This function represents the underlying mathematical relationship between the dependent and independent variables, which is not necessarily easy to deduce by just observing the data set $\mathcal{O}$. The mapping of f can be visualized as a hypersurface in $\mathbb{R}^{n+1}$ that contains every data point in $\mathcal{O}$, and it can be used to predict new not previously observed data.

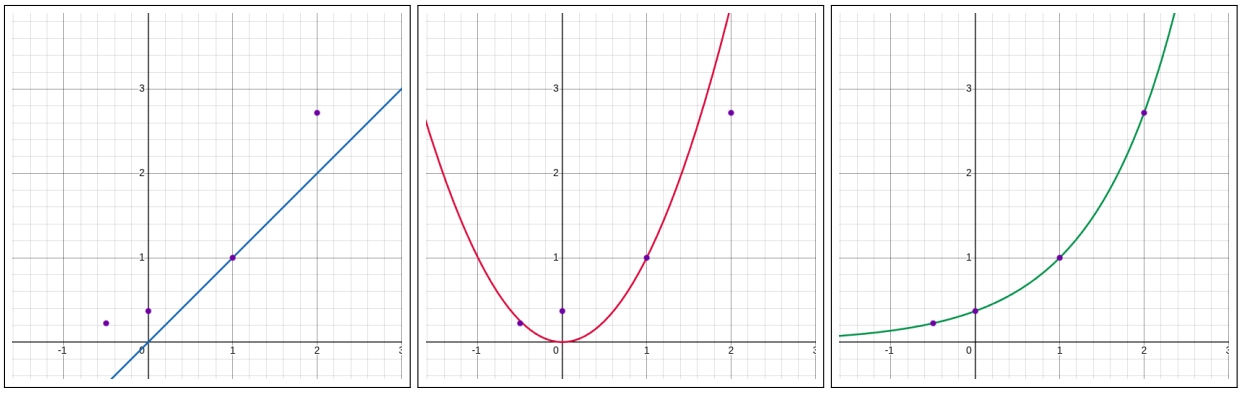


Figure 1. A visualization of three different functions ($f(x) = x$, $f(x) = x^2$ and $f(x) = e^{x-1}$) relative to a set of hypothetical data points in $\mathcal{O}$ (purple dots). The horizontal axis represents the space of independent variables and the vertical axis is the dependent variable space.

Figure 1 serves as a hypothetical representation of what a symbolic regression process might look like. In it, three different models are trialed: a linear one, a quadratic one and an exponential one. The rightmost (exponential) fits to every data point and will therefore be the output f of our algorithm. We would then use f to predict future data knowing that if the model fits well to the existing data, it's likely that it can predict new data with good precision. In other words, if we wanted to deduce where

a new data point $(\mathbf{x}^*, y^*)$[2] outside of $\mathcal{O}$ would land, our best guess would be the point of the curve (hypersurface) that corresponds to $f(\mathbf{x}^*)$.

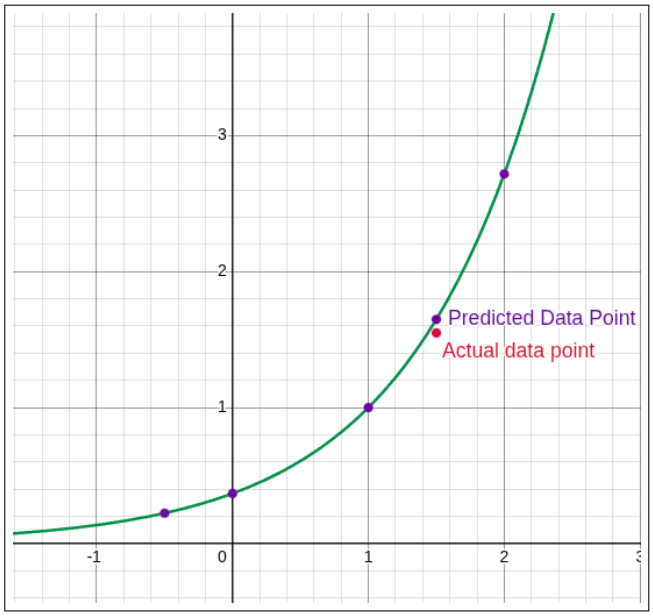


Figure 2. Hypothetical example of what a prediction of a data point might look like with respect to the actual data observed *a posteriori*.

The power of SR — as well as the difference with respect to more classical *regression methods*[3] — comes from the fact that it generates mathematical models purely from patterns in observed data without the need for any prior knowledge about the system that generates said data. For example, *linear regression* aims to optimize the parameters of a linear model to fit the observed data as well as possible, but a successful SR algorithm directly generates a fitting model (linear or not) ”from scratch”. This is a crucial characteristic of symbolic regression as it eliminates potential biases introduced by human assumptions. Moreover, it is not constrained by gaps in prior knowledge about the system under study.

2.2 Symbolic regression’s biggest challenges

Needing so little prior knowledge about the observed data is what makes symbolic regression really powerful, but it is also what makes it so hard to execute successfully. The main problem is that there is no real initial reference around which we can build our desired model, so finding the one among an infinite number of possibilities that fits perfectly to our data is a truly enormous challenge. It falls into the category of **NP-hard** problems, meaning that there are no known algorithms that guarantee that a satisfactory mathematical model will always be found in polynomial time. This means that some form of heuristic[4] should be introduced in order to make the problem more tractable.

[2]The vector notation for the independent variables is kept for consistency purposes. Since this example is in $\mathbb{R}^2$, the point can be simply written as (x^*, y^*).

[3]Set of data analysis methods that use statistical processes to estimate relationships between variables.

[4]A strategy which allows us to find ”good enough” solutions to a hard problem in an efficient way. It is usually a rule of thumb to be followed, a cost function to be minimized, or a constraint on the problem we are trying to solve.

The first and most intuitive constraint to consider is introducing a tolerance into the fitness metric. This allows the mathematical model to diverge slightly from the observed data, which can often lead to a more generalizable solution to a symbolic regression problem. It also allows us to work with *noisy*[5] data, which is present in pretty much every practical study of a system.

While error tolerance aids in simplifying the task of symbolic regression and makes it more flexible in terms of the observed data that we input, it is not strong enough to guarantee a satisfactory result. The more enhanced version of this constraint is the **fit-complexity tradeoff**, which — as its name suggests — allows us to sacrifice the precision of a model to gain simplicity and the ability to generalize it easier. This is a crucial concept for SR, especially when working with a large and/or noisy collection of data. The complexity of an expression can be measured in different ways, most of which use *expression trees* (section 2.3) to calculate a natural number that represents how ”complicated” said expression is.

In essence, adding a lot of terms to an expression makes it more detailed and therefore a better fit to the data when the collection of data points is large and noisy, but it can easily lead to overfitting. In addition, experience tells us that good prediction models in science very rarely come from incredibly (syntactically) complex equations. On the other hand, if we choose a model that's too simple and loosely fits the data, we risk ignoring crucial insight about the relationship between variables.

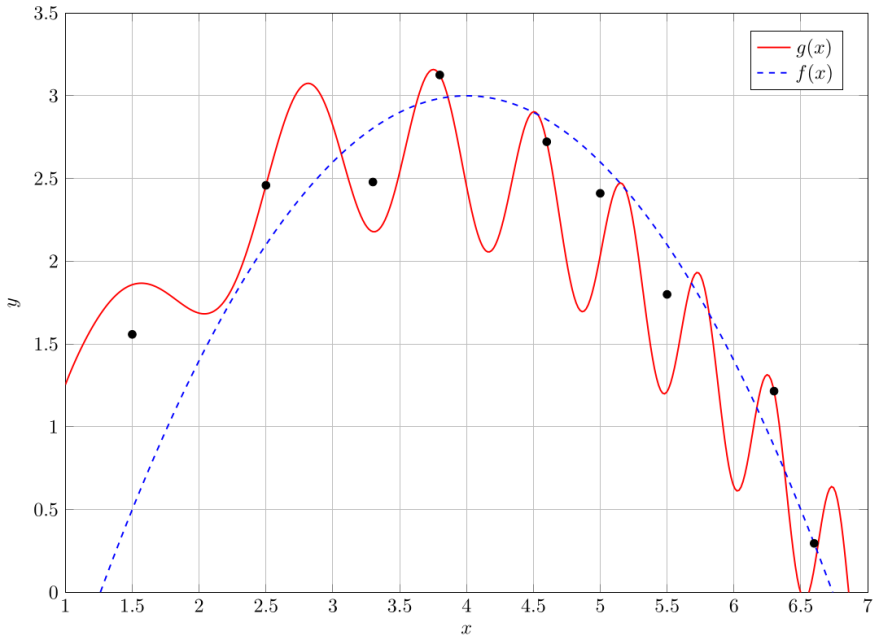


Figure 3. Example of the fit-complexity tradeoff. $g\left(x\right) = 0.5 \cdot \sin\left(x^2 + x^{-3}\right) + \log_{10}\left(x^3\right) - 0.2x^2 + x$ is a better fit to the data, but $f\left(x\right) = -0.4\left(x - 4\right)^2 + 3$ is a much simpler and tractable expression. In this case, $f(x)$ is the preferred model.

To summarize, symbolic regression algorithms must be capable of efficiently generating and trialing a great number of mathematical expressions while balancing their fitness and complexity in order to give a result that's reasonable, tractable and generalizable.

[5]Subject to an error or degree of divergence with respect to the real error-free data.

2.3 Expression trees

Mathematical expressions are often stored and represented using **expression trees**. An *expression tree* is a graph-like data structure that captures the hierarchical nature of a mathematical expression. It is typically implemented as a *binary tree*[6], where internal nodes represent mathematical operators or functions, and leaf nodes represent variables or constants.

- **Operator nodes** (such as $+$, $-$, $*$, $/$) have two child nodes, corresponding to the two sub-expressions on which the operator acts.
- **Function nodes** (such as sin, cos, log) have a single child node, representing the argument of the function.

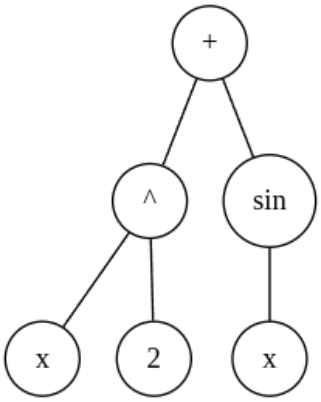


Figure 4. The expression tree corresponding to $x^2 + \sin(x)$.

Expression trees can be read by recursively visiting the nodes in one of the following ways:

- **Pre-order:** Read parent node $\rightarrow$ read left child $\rightarrow$ read right child. A pre-order read of the tree in figure 4 outputs "$+ \; \hat{} \; x \; 2 \;$ sin $\; x$", also known as *prefix* notation.
- **In-order:** Read left child $\rightarrow$ read parent node $\rightarrow$ read right child. An in-order read of the tree in figure 4 outputs "$x \; \hat{} \; 2 \; + \;$ sin $\; x$", also known as *infix* notation.
- **Post-order:** Read left child $\rightarrow$ read right child $\rightarrow$ read parent node. A post-order read of the tree in figure 4 outputs "$x \; 2 \; \hat{} \; x \;$ sin $\; +$", also known as *postfix* notation.

Infix notation is the most commonly used format for writing mathematical expressions, where binary operators are placed between the two sub-expressions they act on. While this notation is easier for humans to read, it requires the use of parentheses to clarify the order in which operations are performed. In contrast, prefix and postfix notations — though harder to read — do not require parentheses, as the position of the operator (before or after the operands, respectively) defines the order of operations [1].

As mentioned in section 2.2, expression trees can be used to measure and control the complexity of mathematical models. Many different metrics can be implemented, the most common of which are:

[6]Hierarchical data structure in which each node has at most two children, referred to as the left child and the right child.

- **Tree length:** It's the total number of nodes of the corresponding expression tree. It directly reflects the length of the expression and does not distinguish the different elements that make it up. The expressions x^2+x and $\sin(\cos(\frac{1}{x}))$ have the same complexity (5) according to this metric, which could lead to confusion, since the latter could be considered more complex in the general sense of the word.

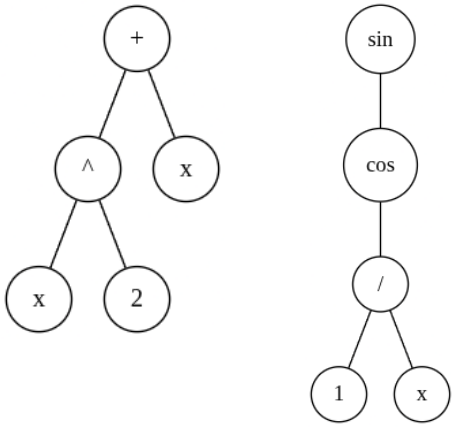


Figure 5. The trees corresponding to $x^2 + x$ and $\sin(\cos(\frac{1}{x}))$. Both trees have the same number of nodes (5) and, therefore, the same length.

- **Tree depth:** Also referred to as **tree height**, it's the distance of the *shortest path*[7] between the root of the expression tree and the leaf that is the furthest away from it. It is calculated by counting the number of nodes that form said shortest path and can be intuitively thought of as the number of "levels" that an expression tree has. The more levels a tree has, the higher the number of nested operations in the expression, and the more complex it is.

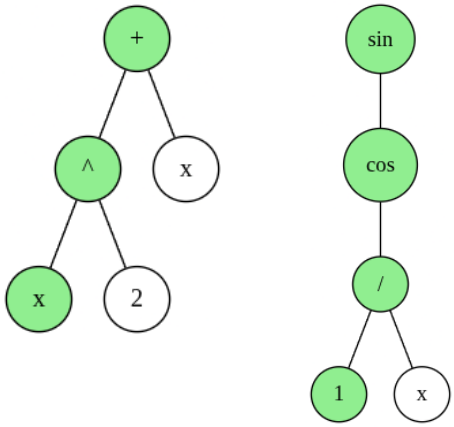


Figure 6. Comparison of the depths of the trees corresponding to $x^2 + x$ and $\sin(\cos(\frac{1}{x}))$. The colored nodes represent the shortest path between the root and the furthermost leaf of each tree.

[7]Given the nodes v_1 and v_2, the shortest path is the smallest possible set of nodes that connect v_1 to v_2. Both v_1 and v_2 are also part of the shortest path between them.

- **Operator penalty:** It associates a certain penalty for every operator (inner) node there is and adds all penalties to determine the complexity of a tree. The weight of the penalty can vary depending on the type of operator (for example, ”+” is a simpler operator than ”sin” or ”cos”), or it can be uniform across all operators. Normally, a uniform distribution means that every operator gets a penalty of 1, in which case the complexity of the expression is equal to the number of operators it has.

Other more specific and sophisticated complexity metrics, like counting the number of unique symbols or the number of nonlinear terms in an expression, can also be used. The *Bayesian Machine Scientist* model uses the *description length* (a more advanced concept coming from information theory) of an expression to measure its complexity and balance it with its fitness [3].

Most symbolic regression algorithms explore candidate expressions by modifying expression trees or generating entirely new ones following some set of rules or *grammars*. For instance, BMS uses a *Markov Chain Monte Carlo* (MCMC) method with a set of three different tree modification operations to progressively evolve existing expression trees into new candidates. These operations can often become computationally costly, since the number of modification steps needed to convert one tree into a new one (often referred to as *edit distance*) that is similar can be higher than expected. BMS is also prone to getting ”stuck” at expressions that locally minimize the description length and therefore fail to explore the mathematical expression space properly.

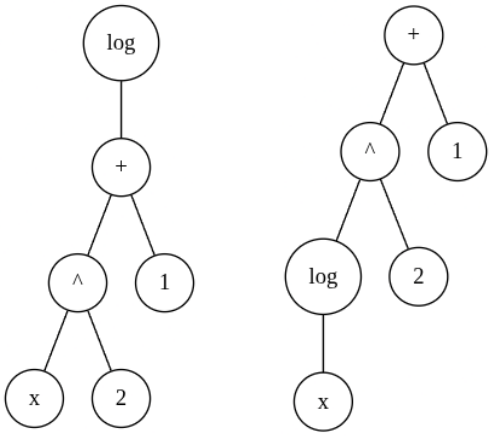


Figure 7. The trees for $\log(x^2 + 1)$ and $\log(x)^2 + 1$ (left and right respectively). The edit distance between the two is higher than expected due to the cost of modifying the root and moving the ”log” node in between the ”x” and ”^” nodes.

As a reminder, our goal will be to determine wether HVAE is a faster and/or more reliable model for generating plausible mathematical expressions for symbolic regression.

2.4 Hierarchical Variational Autoencoder

The **Hierarchical Variational Autoencoder for symbolic regression** is a recent machine learning model that implements a novel way of encoding and decoding mathematical expressions into

continuous $\mathbb{R}^n$-like spaces. HVAE isn't the first *variational autoencoder*[8] (VAE) model proposed for symbolic regression, but the results presented in [1] suggest that it outperforms other known autoencoders. We will not focus on the technical design of the expression encoder and decoder that HVAE implements. This subsection serves as an overview of the fundamental characteristics of HVAE.

We can visualize HVAE as two separate black boxes, one that's capable of encoding expression trees into points z of an n dimensional latent space (*Encoder*), and one that can take a point from said latent space and decode it into an expression tree (*Decoder*). The functions that these black boxes carry out internally will not concern us during the course of this study. Instead, we will purely focus on evaluating their ability to efficiently and reliably encode/decode mathematical expressions. The decoder will be our main focus, since it is able to decode new, previously unvisited points into new, previously unseen mathematical expressions.

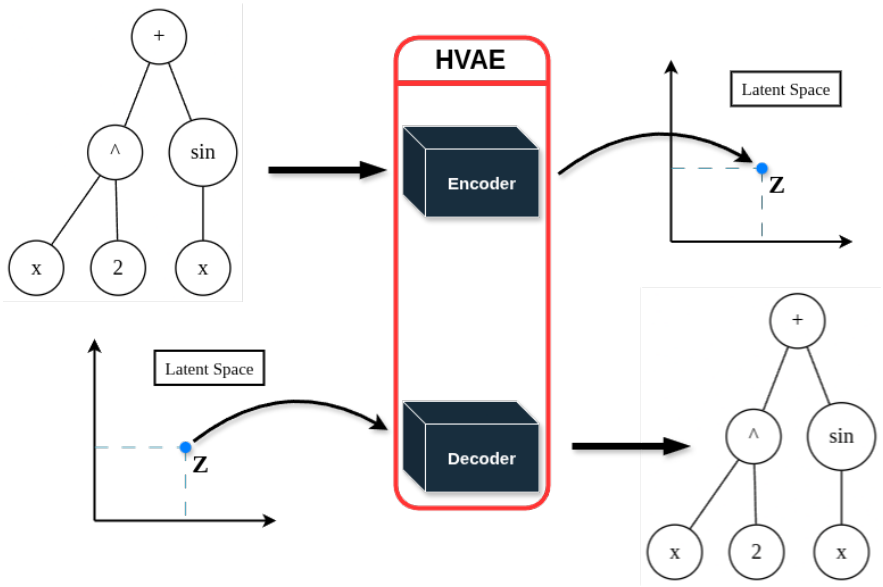


Figure 8. Simple illustration of the architecture of HVAE.

HVAE essentially establishes a mapping between mathematical expressions and points in a continuous space. If this mapping is well learned, we can expect similar expressions to correspond to nearby points within that space. In other words, the latent space will have a coherent structure that can be exploited by search algorithms. Moreover, if we define a function that evaluates the quality of an expression associated with a point in the latent space, the symbolic regression task can be reduced to searching for the maximum/minimum of said function over a continuous domain (a well established problem in algorithmics).

The "smoothness" of the latent space of HVAE is demonstrated in [1] with a *linear interpolation* experiment. It consists of encoding two expressions, A and B, and defining a straight between the two corresponding latent space points. The encoding of the two expressions by z_A and z_B, respectively. Then, the parametrized straight $z^*(\alpha) = \alpha \cdot z_A + (1 - \alpha) \cdot z_B$ is defined, with $\alpha \in [0, 1]$ being the parameter of said straight. The objective is to decode points along $z^*(\alpha)$ for different values of α and

[8] Generative model that learns to encode input data into structured latent spaces from which it can later decode new, similar data

observe the changes in the syntax and complexity of the expressions. Ideally, no big and sudden changes in the expressions should be observed along $z^*(\alpha)$. An important characteristic of HVAE is that it does not output syntactically incorrect mathematical expressions. As seen in figure 9, this is not necessarily the case with other VAE models.

α	HVAE	GVAE	CVAE
Expression A	$c \cdot \cos c + \frac{c}{x} + \frac{x}{\sin x}$	$c \cdot \cos c + \frac{c}{x} + \frac{x}{\sin c}$	$c \cdot \cos c + \frac{c}{x} + \frac{x}{\sin c}$
$\alpha = 0$	$c \cdot x + \frac{c}{x} + \frac{x}{\sin x}$	$c \cdot \cos c + \frac{c}{x} + \frac{x}{\sin c}$	$c \cdot \cos c + \frac{x}{x^x} \cdot \sin x$
$\alpha = 0.25$	$c \cdot \sin c + \frac{c}{x} + x$	$c + \frac{c}{\cos c} + c \cdot x$	$\frac{c}{\cos c} + c - \frac{x}{c} \cdot \sin x$
$\alpha = 0.5$	$\frac{c \cdot \sin c}{c} + x$	$c + \frac{x}{c^{\cos c}} \cdot x$	$c - \cos c - \frac{x}{x} \cdot \sin c$
$\alpha = 0.75$	$\frac{\sin(x - \sin c)}{\frac{x}{\sin(x-c)}}$	$x + c \cdot \frac{c}{x} - x$	$\cos(x(c) - c \cdot$
$\alpha = 1$	$\frac{\sin(x-c)}{x}$	$\frac{\sin(x-c)}{\frac{x}{\sin(x-c)}}$	$\frac{\sin(x-c)}{\frac{x}{\sin(x-c)}}$
Expression B			
Expression A	$x - \sin(c \cdot x)$	$x - \sin(c \cdot x)$	$x - \sin(c \cdot x)$
$\alpha = 0$	$x - \sin(c \cdot x)$	$\frac{x}{\sin(c \cdot x)}$	$x - \sin(c \cdot x)$
$\alpha = 0.25$	$x \cdot \sin c - \sin(c \cdot x)$	$\frac{x}{\sin(c \cdot x)}$	$x - \cos(c \cdot x)$
$\alpha = 0.5$	$c \cdot \sin x + x$	$c + \cos c$	$\frac{c}{\cos} - \cos \cdot c) - c$
$\alpha = 0.75$	$c \cdot \cos \frac{x}{c} + c$	$c \cdot x \cdot \frac{\cos(\frac{x}{c})}{c}$	$c \cdot x \cdot \frac{\cos(\frac{c}{x})}{c}$
$\alpha = 1$	$c \cdot x \cdot \cos \frac{x}{c} + c$	$c \cdot x \cdot \cos(\frac{x}{c}) + c$	$c \cdot x \cdot \cos(\frac{x}{c}) + c$
Expression B	$c \cdot x \cdot \cos \frac{x}{c} + c$	$c \cdot x \cdot \cos(\frac{x}{c}) + c$	$c \cdot x \cdot \cos(\frac{x}{c}) + c$

Figure 9. Table extracted from "*Efficient generator of mathematical expressions for symbolic regression.*" [1] showcasing the linear interpolation experiment done with three different VAEs. The expressions in red are syntactically invalid.

Having a well-structured latent space is crucial for the efficiency of symbolic regression algorithms. Recall from 2.3 that converting an expression into a new, syntactically similar one can be more difficult than expected due to the cost of tree modification operations. A well-structured latent space allows us to skip the tree modification step. Instead, if we have an expression A and want to generate a similar one, we can simply select a latent space point that is close to the encoding of A, and decode it into a new expression. We will conduct experiments to further study and characterize the structure of the latent spaces generate by HVAE (3.1).

The dimension of the latent space will be set at 32 (i.e. the space will be $\mathbb{R}^{32}$) for the duration of this study, as it is suggested to be an optimal latent space size in [1]. This is a low number of dimensions in the context of variational autoencoders for symbolic regression, which is a particular point of interest since lower dimensional spaces are easier to explore. Mežnar and his team show that HVAE outperforms other VAEs in many aspects, regardless of the reduced dimensionality of the latent space.

The practical implementation of HVAE is done in Python, which will be the programming language used in this study. The model must be trained with a collection of pre-defined mathematical expressions. We will not modify the training parameters set by the creators of HVAE. The training sets will contain 4×10^4 expressions, and the model will be trained for 20 epochs.

3 Results

A total of five different experiments will be conducted in order to quantitatively evaluate the performance of HVAE as a generator for mathematical expressions. The experiments will be split into two groups:

- **Latent space characterization experiments**, aimed at gaining insight into the structure of the encoded latent spaces of HVAE.

  This category will include three experiments: random sampling, random walks and encode-decode reconstruction. The main goal will be to learn how expressions are distributed inside the latent space and how reliable and consistent HVAE is in generating them.

- **Symbolic regression experiments**, aimed at directly testing the performance of HVAE in symbolic regression scenarios.

  Two different SR algorithms will be implemented using HVAE as an expression generator. The tests will be run with data generated from random expressions decoded from the latent space, as well as data from the *Nguyen Benchmark* for symbolic regression [5].

3.1 Latent space characterization experiments

We begin the quantitative evaluation of HVAE by conducting three different space characterization experiments. In this section, we will not directly evaluate the quality of the generated expressions in the context of symbolic regression. Our focus will be to empirically study how expressions are distributed within the latent space. Firstly, we randomly sample points from the latent space and observe how different positions in the latent space affect the characteristics of the decoded expressions (3.1.1). We then run different random walks inside the latent space that aim to further study its "smoothness" (3.1.2). Finally, we evaluate how consistently different sets of expressions are mapped to and recovered from the latent space (3.1.3).

All experiments in this section will be conducted with HVAE trained with 32-dimensional spaces with two different expression sets: NG1-7 and BMS-NG. The first one is used by the creators of HVAE [1] for experiments on the *Nguyen Benchmark* [5], the expressions in it have a single variable "x" and contain operators and functions from the symbol library $\{+, -, *, /, \sin, \cos, \log, \exp, \text{sqrt}, \hat{}2, \hat{}3, \hat{}4, \hat{}5\}$.

The set BMS-NG is an extension to NG1-7, where expressions can also contain constant parameters denoted by "C" and more possible exponentiations. The symbol library for BMS-NG is $\{+, -, *, /, \hat{}, \sin, \cos, \log, \exp, \text{sqrt}\}$. Notice that the "^" operator in does not precede natural numbers like in NG1-7, which allows for expressions with non-natural exponents (like x^x or C^x) to exist. The BMS-NG set will be of particular interest in 3.1.3, but will also be used in the rest of the experiments.

3.1.1 Random *l-space* sampling

We choose the points z by sampling a number from a normal distribution $N(0, \sigma^2 \cdot I)$ for every coordinate of a point. Since we work with $\mathbb{R}^{32}$ as a latent space, sampling the coordinates this way is equivalent to sampling a random vector from $N(\mathbf{0}, \sigma^2 \cdot I_{32})$. In other words, the points will be

sampled from a multivariate Gaussian distribution with variance σ^2 centered around the origin of $\mathbb{R}^{32}$. We also take into account the distance from the origin to each point by taking the *Euclidean norm* of the corresponding vector. We aim to empirically observe the complexity distributions for different values of σ and determine how the distance from origin affects the expression complexity.

We run the sampling experiment with the standard deviation values of $\sigma = 0.5,\ 1,\ 1.5,\ 2$, and sample 10^4 latent space points for every run. We use the *PyTorch* library's `randn()` method to generate a 32-component vector sampled from $N(\mathbf{0}, I_{32})$ and then scale it by σ, which results in a vector $\mathbf{z}$ sampled from $N(\mathbf{0}, \sigma^2 \cdot I_{32})$. We calculate the Euclidean norm of $\mathbf{z}$ as

$$\|\mathbf{z}\|_2 = \sqrt{\sum_{i=1}^{32} z_i^2}$$

where z_i are the components of $\mathbf{z}$. We use it to keep track of the distance at which $\mathbf{z}$ is situated with respect to the origin of the latent space.

We then decode $\mathbf{z}$ into a mathematical expression using HVAE. The complexity of the expressions will be calculated by counting the number of operators each one contains (uniform operator penalty, section 2.3). We plot the modulus-complexity correspondence for every sampled point, as well as a complexity histogram that showcases the distribution of complexities inside the latent space region from which points are sampled. In addition, the expressions will be ranked based on how many times each one was sampled during the experiment.

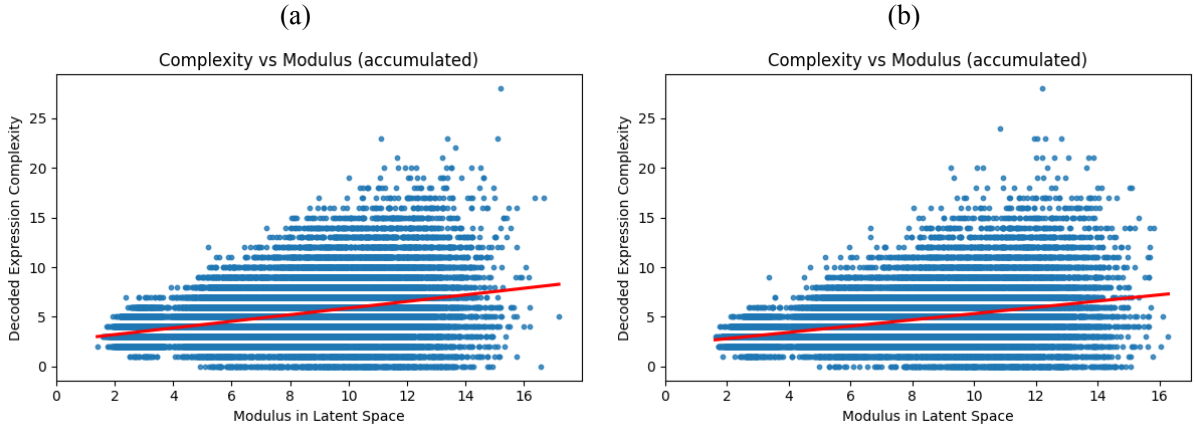


Figure 10. Scatter modulus-complexity plots with added least-squares lines to showcase the overall tendency of the relationships. (a) HVAE trained with NG1-7. (b) HVAE trained with BMS1-8.

Figure 10 shows that, overall, simpler expressions tend to be more concentrated near the origin (center) of the latent space, and more complex models become available as we move further away. While the range of expression complexities increases as the Euclidean norm of $\mathbf{z}$ grows, we continue to recover really simple expressions (complexity 0 or 1) throughout the entire domain of the experiment. In particular, with HVAE trained with NG1-7, an expression with complexity 0 can only be x (no operators). We can see from f. 10 (a) that it is recovered over a large range of distances, suggesting that the same expression can be encountered in different regions of the latent space, which is not ideal for the efficiency of SR algorithms.

The rate of increase of the complexities remains stable over the course of the experiment, with no big spikes or pits in complexity occurring in any particular region. This suggests that the modulus-complexity relationship remains consistent in the regions around the center of the latent spaces.

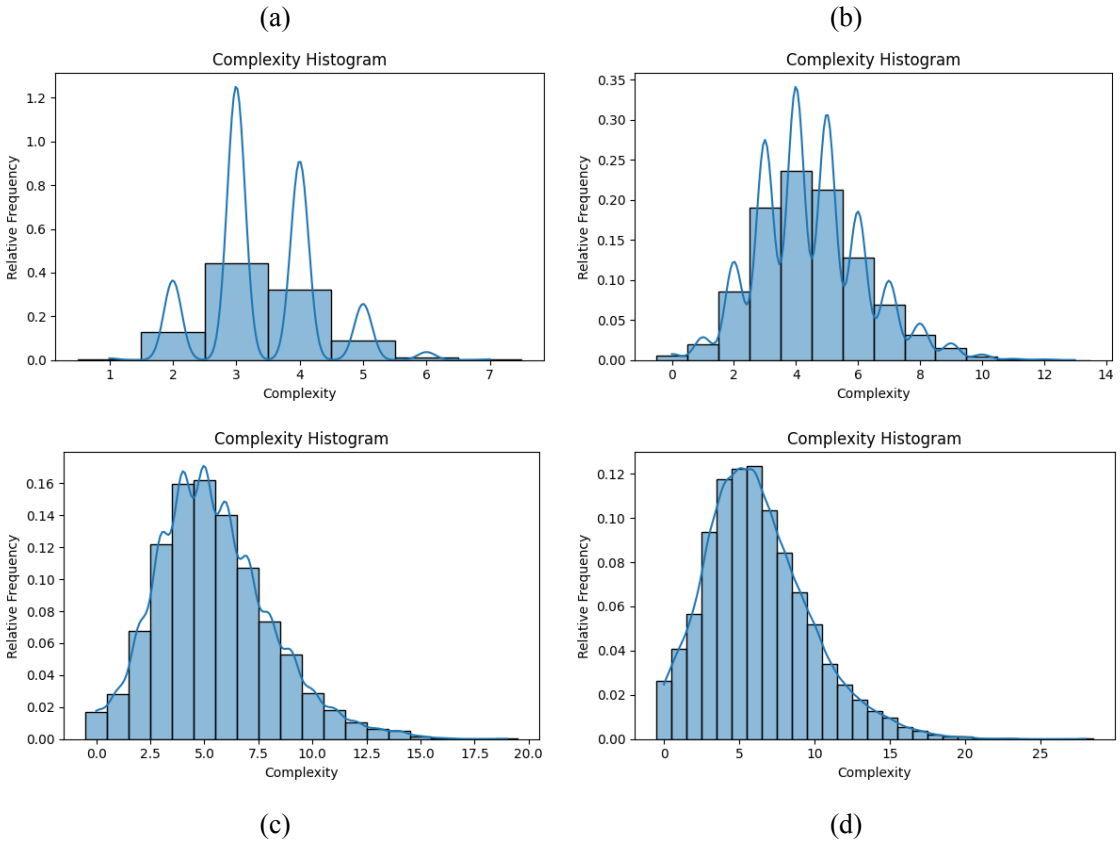


Figure 11. Complexity distributions for (a) $\sigma = 0.5$, (b) $\sigma = 1$, (c) $\sigma = 1.5$ and (d) $\sigma = 2$. Training set: NG1-7.

Figures 11 and 12 show how the expression complexity distribution evolves as we increase the standard deviation (spread of points) of the sampling around the origin of the latent space. The continuous blue lines in the figures are Kernel Density Estimations (KDE) of the distributions. They are displayed as a reference to highlight how the spread of the distributions changes for different values of σ.

As the sampling standard deviation increases from 0.5 to 2.0, the resulting expressions show a clear shift toward higher complexities and greater diversity. At a low standard deviation (0.5), the model mainly generates simple expressions with a narrow distribution, indicating a low degree of variability. As the standard deviation increases, the distributions become wider, reflecting a balance between complexity and diversity. At the highest standard deviation (2.0), the distribution becomes right-skewed, with a long tail of high-complexity expressions, suggesting that the model is sampling from increasingly distant and less structured regions of the latent space. This trend highlights once more that regions closer to the origin yield more reliable but repetitive outputs, while higher deviations promote more variety at the potential cost of syntactic validity.

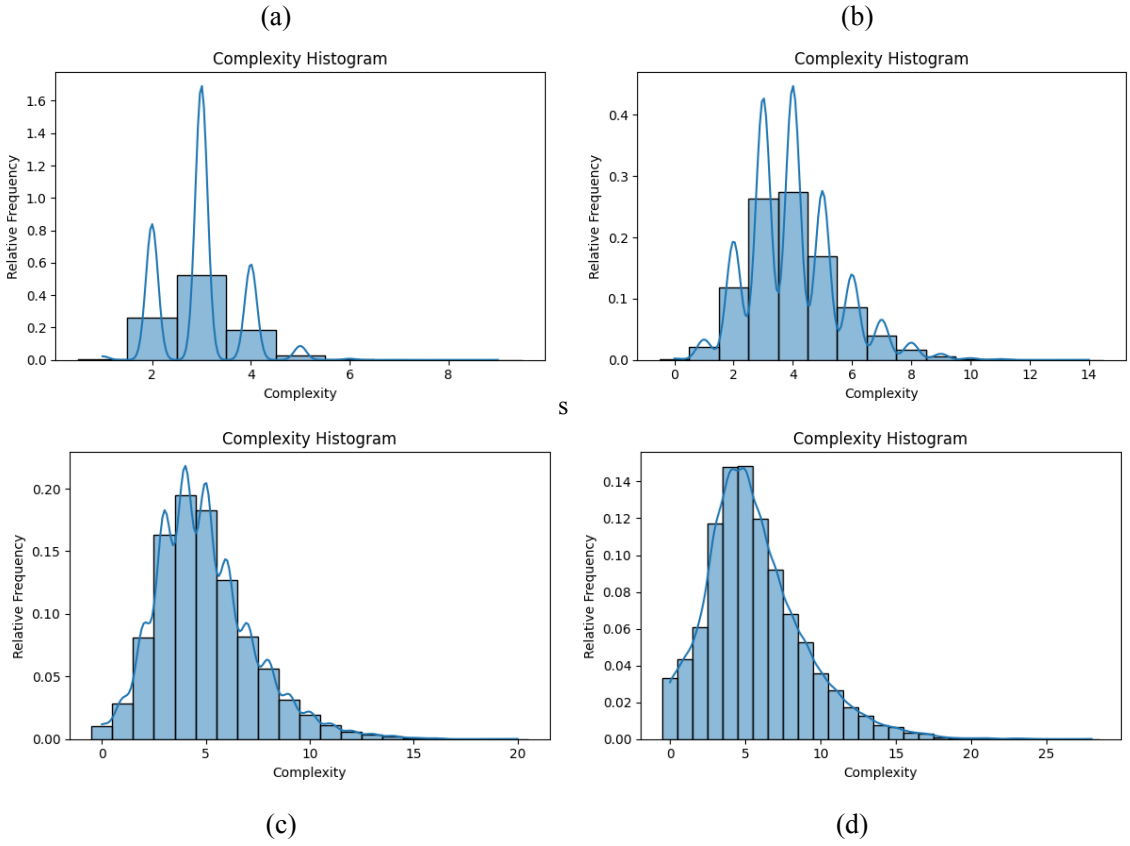


Figure 12. Complexity distributions for (a) $\sigma = 0.5$, (b) $\sigma = 1$, (c) $\sigma = 1.5$ and (d) $\sigma = 2$. Training set: BMS-NG.

In addition, we can observe that the shift in the distribution between $\sigma = 0.5$ and $\sigma = 1.5$ with HVAE trained with BMS-NG happens slower than with HVAE trained with NG1-7. This indicates that expressions are simpler and more repetitive around the origin of the latent space obtained from BMS-NG, which is an unexpected result since the set allows for more operations than NG1-7.

3.1.2 Random walks

The following part of this study will be focused on the examination of the ”smoothness” of the latent spaces created by HVAE. We do so by implementing a *random walk* model within the spaces and measure the evolution of the decoded expressions along the steps. Our goal is to explore spaces by taking a very small, random increment at each step of the walk. Similarly to the example in figure 9, we expect to notice a gradual, smooth transition in the decoded expressions when moving between different areas of the latent space.

We implement the random walk as follows:

$$X_t = X_{t-1} + \delta_t \tag{1}$$

where X_t represent the position in a space at step t, and δ_t is the random change in position applied

at step t. In our case, X_t is a point in the HVAE latent space, while δ_t is an increment sampled from $N(0, \sigma_{\delta_t}^2 \cdot I_{32})$. The starting point of the walks will be set to $X_0 = \mathbf{0}$, i.e. the origin of the latent space. We can therefore re-write equation 1 as:

$$X_t = \sum_{i=1}^{t} \delta_t \tag{2}$$

Notice that, since δ_t is chosen from a symmetrical distribution centered at $\mathbf{0}$, the expected value for δ_t ($\mathbb{E}[\delta_t]$) is $\mathbf{0}$. Therefore, due to the linearity of the expected value, we have:

$$\mathbb{E}\left[X_t\right] = \mathbb{E}\left[\sum_{i=1}^{t} \delta_t\right] = \sum_{i=1}^{t} \mathbb{E}\left[\delta_t\right] = \mathbf{0}$$

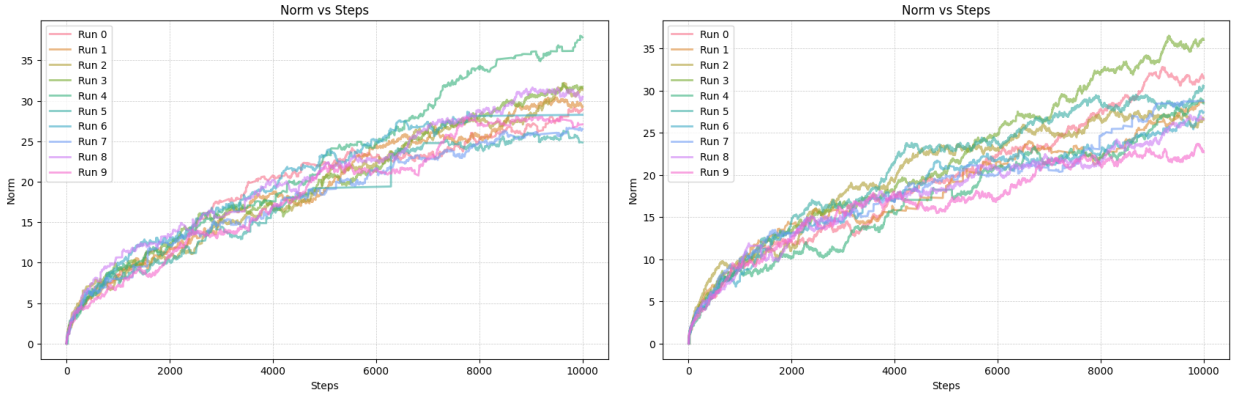
which is a well-known property of random walks that can be easily misinterpreted. The average position of a symmetric random walk is $\mathbf{0}$, but that doesn't mean that there won't be any net movement after t steps. In fact, on average, the distance $\|X_t\|_2$ traveled at step t is proportional to $\sqrt{t}$ given a symmetrical distribution of steps (figure 13). We use the random walk model to progressively drift further away from the origin of the latent space in random directions while keeping track of the evolution of the decoded expressions.

We run sequences of 10 random walks for 10^4 steps each, and set the standard deviation for the increments to $\sigma_{\delta_t} = 5 \times 10^{-2}$. The initial vector $\mathbf{z}_0$ for every walk is set at $\mathbf{0}$, and we then use the same method as in section 3.1.1 to sample an increment vector δ_t from $N(\mathbf{0}, \sigma_{\delta_t}^2 \cdot I_{32})$ at step t. The increment is cumulatively added to $\mathbf{z_0}$, thus obtaining

$$\mathbf{z}_t = \sum_{i=1}^{t} \delta_t, \quad 1 \leq t \leq 10^4$$

The modulus of $\mathbf{z}_t$ is once again calculated as the Euclidean norm at every step of the walk, and the vector is decoded into a mathematical expression using HVAE.

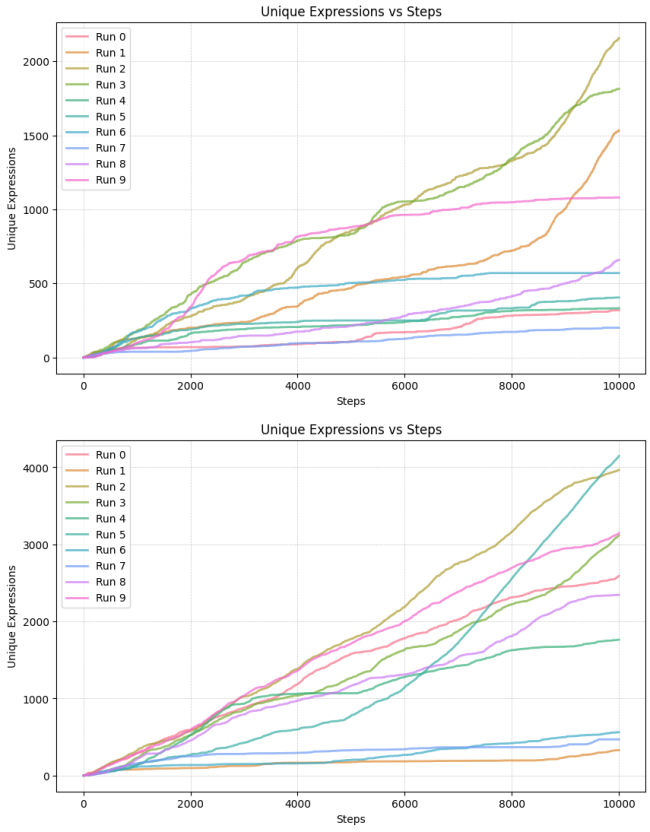
We keep track of the number of unique expressions visited during each walk (figure 14). An expression is considered to be unique if the exact structure of the corresponding expression tree has not been decoded in a previous step of the walk. For reference, $cos(x)+x$, $x+cos(x)$ and $x+cos(x+x-x)$ are all considered different expressions during this experiment. We also keep track of the total number of steps spent at (visits of) every unique expression during the walk.



(a) (b)

Figure 13. Showcase of the evolution of the traveled distance with respect to the number of steps during every random walk in this experiment. (a) HVAE trained with NG1-7. (b) HVAE trained with BMS-NG.

(a)



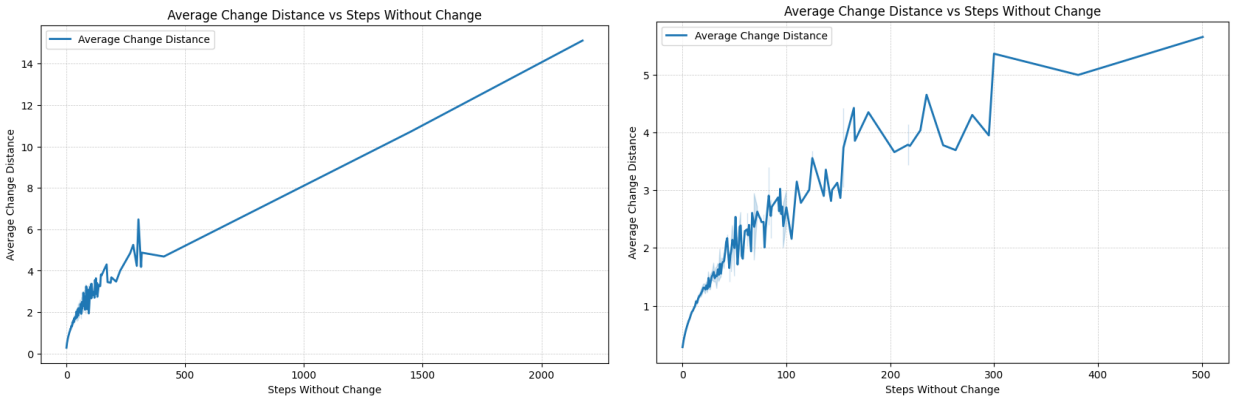
(b)

Figure 14. Number of unique expressions visited for each of the 10 random walks. (a) HVAE trained with NG1-7. (b) HVAE trained with BMS-NG.

In figure 14, the number of unique expressions plateaus in several runs, indicating that the model can wind up decoding redundant expressions during exploration. This suggests limited diversity in the corresponding areas of the latent space; however, there are runs that show a greater rate of increase and reach higher numbers of unique expressions. This indicates that not all regions of the latent space have an equal density of unique expressions.

In 14 (b), we generally observe a steeper and more consistent increase in unique expressions than in 14 (a), with some runs reaching over 4×10^3 unique decodings. This suggests that the second configuration of walks explores the latent space more effectively, containing regions that correspond to a broader range of syntactically distinct expressions. The difference between the two panels is likely due to the difference in expressions allowed by each training set.

The distance between two **consecutively decoded** expressions A and B ($A \neq B$) is measured by $\|\mathbf{z}_{t_B} - \mathbf{z}_{t_A}\|_2$, where t_A is the first step in which A was most recently decoded, and $t_B > t_A$ is the earliest subsequent step at which a change from A to B is detected. We will refer to this measurement as the *change distance* between A and B. The value $t_{AB} = t_B - t_A$ is the number of steps between A and B. We keep track of the *change distance* and the steps between expressions for every pair of consecutively decoded expressions during the walk. These metrics help us study the sizes of the regions that correspond to specific expressions inside the latent space. The more consecutive steps spent in a region, the bigger it is likely to be.

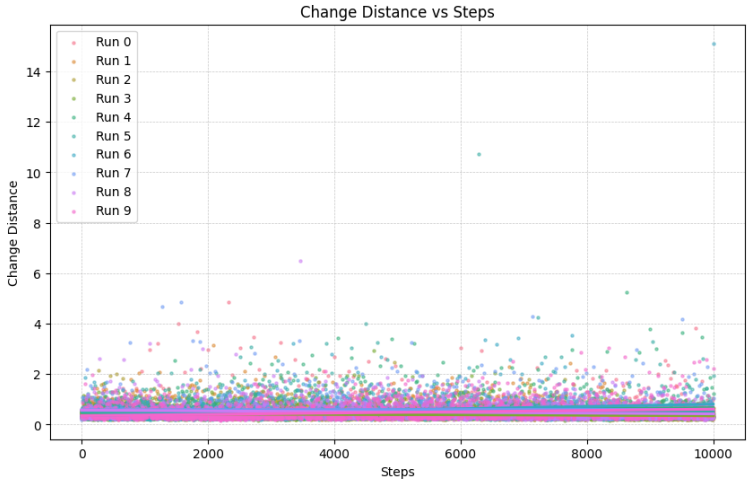


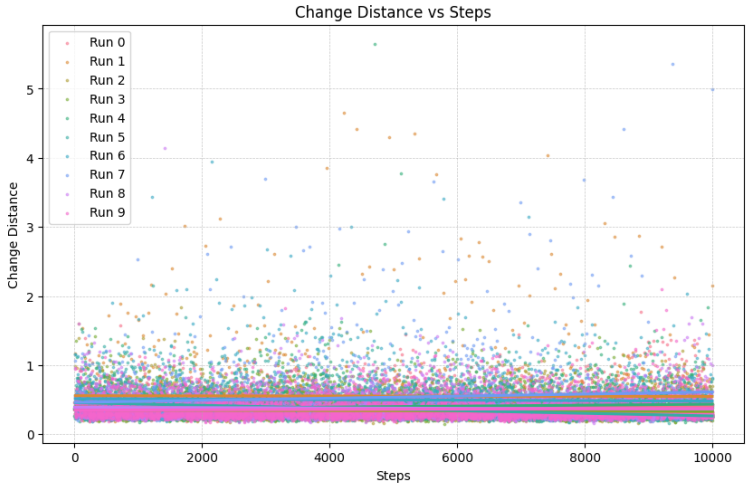
(a) (b)

Figure 15. The average change distance with respect to the number of consecutive steps without a change in the decoded expression. (a) HVAE trained with NG1-7. (b) HVAE trained with BMS-NG.

Spending a larger amount of steps t_{AB} in a single expression region results in traveling a larger distance without a change in the decoded expression. Since we are implementing a random walk with a symmetric distribution of increments, the change distance grows as $\sqrt{t_{AB}}$ (shown in figure 15) on average. We observe that, indeed, there appear to be expression regions with different sizes throughout both latent spaces. In particular, in 15 (a), we can see that there has been a region in which the random walk has spent more than 20 % of its total steps without detecting a change in the decoded expression. This is a clear indicator of the irregularities in expression region sizes within the latent space, which isn't necessarily a bad characteristic since, if overly complex expressions are mapped to smaller regions, they can be more easily avoided. Nonetheless, having a dis-balance as big as what is seen in figure 15 (a) is not desirable since it can seriously reduce the search efficiency of a symbolic regression algorithm.

(a)



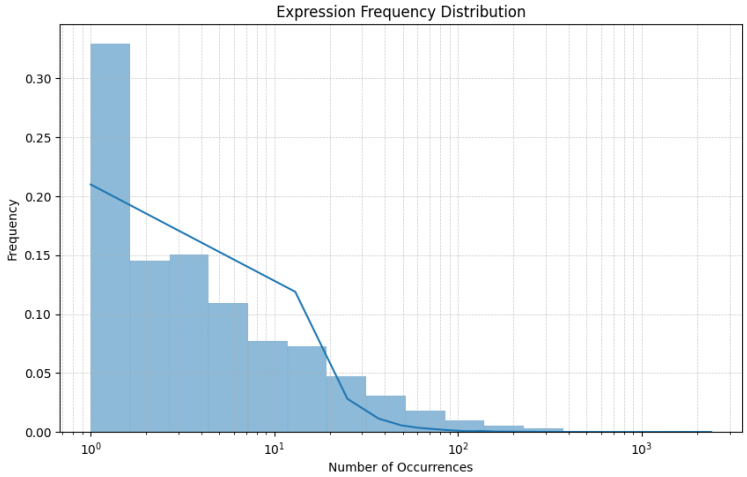


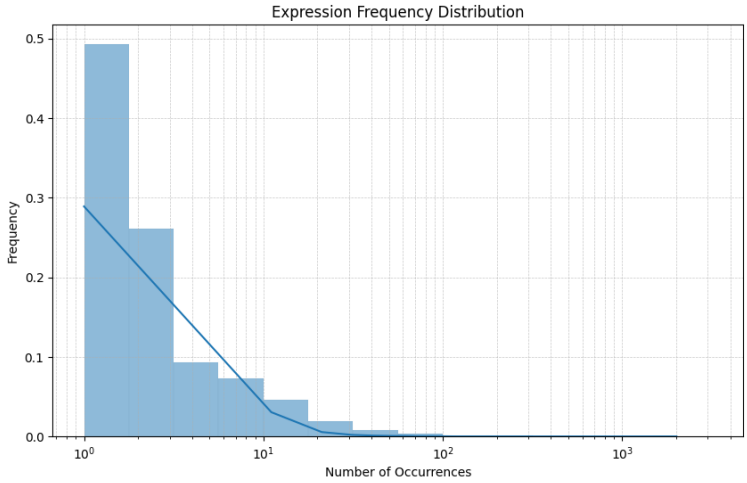
(b)

Figure 16. Showcase of the evolution of the change distance in the latent space with respect to the steps taken in each random walk. (a) HVAE trained with NG1-7. (b) HVAE trained with BMS-NG.

In figure 16 we see that there is not any clear relationship between the measured change distances and the total steps traveled during the walk. This suggests that the sizes of expression regions are evenly distributed within the latent spaces, and smaller regions are far more likely to be found at any step of the walk. We extend the notion of expression region size by counting the total number of visits of every unique expression encountered during a walk and then plot a relative frequency histogram of said number.

(a)



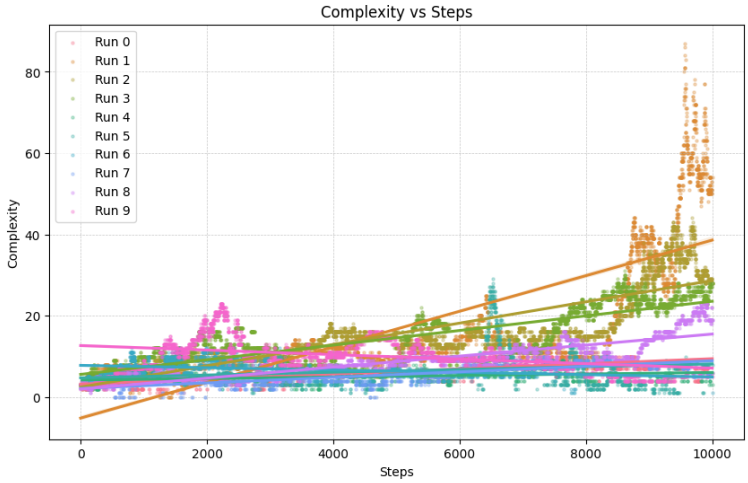


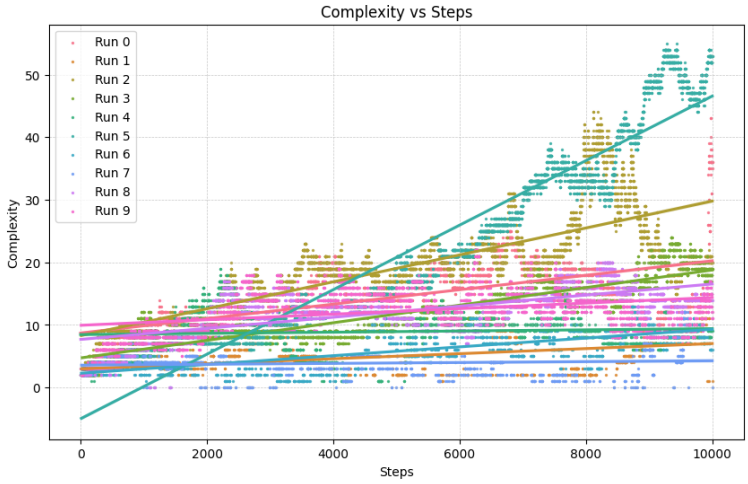
(b)

Figure 17. Relative frequency histograms of the total number of visits of every unique expression across the 10 walks. The horizontal axis is set to a logarithmic scale in order to represent the distributions more compactly. (a) HVAE trained with NG1-7. (b) HVAE trained with BMS-NG.

In figure 17 (a), corresponding to the HVAE trained on NG1-7, the distribution exhibits a pronounced long tail, with several expressions being revisited hundreds or even thousands of times. This indicates a high degree of redundancy in the decoded outputs, suggesting again that the latent space contains large regions that map to the same expression. In contrast, for HVAE trained on BMS-NG (figure 17 (b)), reveals a steeper decay in frequency and a higher concentration of expressions that are visited only once or a few times. This reflects greater decoding diversity and implies that the latent space learned from BMS-NG contains a more diverse structure. Overall, we can conclude that HVAE can contain high redundancy regions inside its latent spaces.

(a)





(b)

Figure 18. Scatter of the complexity vs. steps relationship for all of the random walks. The added straight lines linearly approximate the behavior of each of the walks. (a) HVAE trained with NG1-7. (b) HVAE trained with BMS-NG.

Figure 18 shows us how the complexities of the decoded expressions evolve during the course of each walk. We can see that both configurations of HVAE remain relatively stable and similar during the first half of every walk, with some runs retaining that stability all the way through. However, there clearly are runs which experience a higher increase rate in the second half, reaching a complexity peak of around 50 in 18 (b) and over 80 in 18 (a). Additionally, We can see that one of the runs in 18 (b) experiences a really big spike in complexity in its final steps, which indicates a non-smooth transition between expressions. In figure 18 (a) the walks appear to remain stable for longer, however, higher peaks in complexity are reached. This exposes noticeable inconsistencies in the structure of the latent spaces as we move further away from the origin.

Table 1. The 15 most visited expressions (in descending order) after 10 random walks in both latent spaces (NG1-7 on the left and BMS-NG on the right).

Expression	**Visits** (out of 10^5)	**Expression**	**Visits** (out of 10^5)
$\log(x)$	2386	3	2005
$\log(\log(x))$	2177	$C \cdot x$	829
x	1230	$\log(4)$	747
$\sin(\sin(x))$	1068	$\sqrt{5-5}$	696
$\exp(\exp(\cos(x)))$	495	$\log(\cos(C))$	672
$(\exp(x \cdot \sin(x)))^2$	483	$C \cdot \exp(C)$	661
$\log(\log(\log(x)))$	456	$\log(C^3 - C)$	653
$\left(\frac{\cos(x)}{x} \Big/ \left(\frac{\sqrt{\log(x)}}{\log(x)}\right)^4\right)$	456	$\sqrt{x - C}$	603
$\sin(x)$	399	$\log(x - C)$	595
$\log\left((x-x) - \frac{x \cdot x}{x}\right)$	385	$\log(C - C)$	560
$\frac{\exp(x)}{\cos(\exp(x))}$	363	$\log(C - \log(C))$	451
$\left(\frac{x^2}{x} \Big/ \left(\frac{\sqrt{\sqrt{\log(x)}}}{\sqrt{x}}\right)^4\right)$	359	$\log((C^5)^5)$	449
$\left(\log\left(x + \frac{x}{(x \cdot x)^3} + x\right)\right)^3$	352	$\log\left(\cos\left(\frac{C}{C}\right)\right)$	363
$\left(\frac{\sqrt{x}}{x} \Big/ \left(\frac{\sqrt{\log(x)}}{\log(x)}\right)^4\right)$	351	$3 \cdot x$	362
$x - \left(x^3 \cdot x\right)^4$	338	$\log(2)$	334

Finally, we examine the most frequently visited expressions in table 1, where the trend of HVAE trained on NG1-7 yielding more complex expressions is clearly noticeable. HVAE trained on NG1-7 tends to revisit structurally complex and deep expressions, with multiple instances of highly nested functions, like $\exp(\exp(\cos(x)))$, being observed. On the other hand, HVAE trained on BMS-NG shows a bias toward simpler, constant-based forms. An interesting observation is that the BMS-NG set does not contain natural numbers as independent symbols in an expression (they exclusively appear as exponents), yet a few of the most visited expressions (including the most visited one) have natural numbers as independent elements of the expression. Overall, the expressions decoded with HVAE trained on NG1-7 appear to be more useful. The nesting of functions from NG1-7 and the abundance expressions resulting in just a constant from BMS-NG suggest that HVAE might have some shortcomings when learning how to decode expressions.

3.1.3 Encode-decode reconstruction

The last experiment of the latent space characterization part of this study is focused on determining how reliable HVAE is in encoding and decoding expressions. We want to observe the rate with which HVAE successfully encodes an expression into a latent space vector and then successfully recovers it from the same vector. In this experiment, we select different expression collections on which to test the encode-decode reliability of HVAE, and then calculate the success rate for each trial. In addition, we

will be able to determine wether HVAE is able to *deterministically*[9] associate the encoding and decoding of mathematical expressions.

We initially test HVAE on some of the samplings from the experiment in section 3.1.1. Since those expressions were directly extracted from the latent space, we should expect a reasonably high rate of successful recovery. We then move onto the *Nguyen Benchmark* [5] for symbolic regression, where we will trial HVAE on the first eight expressions of the benchmark (table 2). Finally, we use various expression sets generated by the Bayesian Machine Scientist model trained with some of the Nguyen expressions. The BMS-NG training set is crucial in the context of this experiment, since most expressions generated by BMS contain constant parameters, which can't be generated by HVAE trained with NG1-7. The tests will be run multiple times on every expression set in order to determine wether the outcome is deterministic or not.

Table 2. First eight expressions of the Nguyen Benchmark for symbolic regression.

ID	Expression
NG-1	$x^3 + x^2 + x$
NG-2	$x^4 + x^3 + x^2 + x$
NG-3	$x^5 + x^4 + x^3 + x^2 + x$
NG-4	$x^6 + x^5 + x^4 + x^3 + x^2 + x$
NG-5	$sin(x^2) \cdot cos(x) - 1$
NG-6	$sin(x) + sin(x + x^2)$
NG-7	$\ln(x + 1) + \ln(x^2 + 1)$
NG-8	$\sqrt{x}$

The syntax of expressions 4, 5 and 7 from the Nguyen Benchmark (table 2) needs to be slightly modified due to symbol library restrictions. Neither of the expression sets used to train HVAE for this experiment allow for the exponent "^6" nor the symbol "1" to be generated directly. Therefore, the three previously mentioned expressions are accordingly modified in order to be possible for HVAE to encode and decode them.

$$\begin{array}{ccc} x^6 + x^5 + x^4 + x^3 + x^2 + x & \longrightarrow & x \cdot x^5 + x^5 + x^4 + x^3 + x^2 + x \\ sin(x^2) \cdot cos(x) - 1 & \longrightarrow & sin(x^2) \cdot cos(x) - \frac{x}{x} \\ \ln(x + 1) + \ln(x^2 + 1) & \longrightarrow & \ln(x + \frac{x}{x}) + \ln(x^2 + \frac{x}{x}) \end{array}$$

Ideally, the modified expressions should still be recoverable from the latent space, especially when HVAE is trained with NG1-7. In practice, the expressions on the right in the previous diagram can be passed through an expression simplifier after they are generated. Also, the symbol library can be extended to include the required symbols to directly encode and decode the Nguyen expressions, however, we prioritize using the "NG1-7" training set since it's the one used to run experiments on the Nguyen benchmark in [1].

In order to be able to run the experiment on expressions generated by BMS, we must correctly adapt them to the HVAE notation format. The Bayesian Machine Scientist generates expressions that contain constant parameters, denoted by "a_i" for some $i \in \mathbb{N}$, which all have assigned values upon

[9] To always output the same result given a fixed input.

generation. HVAE uses the symbol "C" as a placeholder for constant parameters in a mathematical expression, and a specific value is assigned to every "C" via an independent process of parameter fitting after the generation. Since we only focus on the actual syntax of the expressions, we substitute every "a_i" generated by BMS with a "C". This results in the abundant repetition of expressions due to the fact that BMS can generate multiple "identical" iterations of an expression with different constant parameter values. We therefore filter all repeated expressions after the transformation from BMS to HVAE notation. Finally, some minor adjustments made to ensure that operators and functions generated by the Bayesian Machine Scientist are correctly translated to the symbol libraries used by HVAE.

We now run the reconstruction experiment on the samplings from section 3.1.1, the expressions from table 2, and the sets generated by BMS. Each distinct expression is encoded and decoded 5 consecutive times in order to empirically check if HVAE achieves a deterministic mapping between the input of the encoder and the output of the decoder. We deem a recovery to be successful if the decoded expression matches the input expression **exactly**. We define Python *set objects*[10] to store and track the decodings associated to each individual input expression. We then measure both the total rate of successful recoveries across all input expressions, as well as the individual rate of success for each one.

Table 3. Results of the reconstruction experiment for HVAE trained on BMS-NG.

Expression set	Number of expressions	Successful recovery rate
BMS-NG-NS	2000	0.7855
BMS-Poly	2045	0.0171
BMS-Log	2255	0740
BMS-Trig	1623	0.00061
BMS-Sqrt	2242	0.0976

Table 4. Results of the reconstruction experiment for HVAE trained on NG1-7.

Expression set	Number of expressions	Successful recovery rate
NG1-7-NS	2000	0.903
Nguyen1-8	8	0.25

The expression sets for the experiment are:

- **BMS-NG-NS**: The 2000 most frequently decoded expressions from section 3.1.1 with HVAE trained on BMS-NG and sampled from $N(0, I_{32})$.
- **NG1-7-NS**: The 2000 most frequently decoded expressions from section 3.1.1 with HVAE trained on NG1-7 and sampled from $N(0, I_{32})$.
- **BMS-Poly**: Expressions generated by BMS trained with mainly polynomial expressions with constant parameters from the Nguyen dataset.

[10]Data structure that doesn't allow the storing of two identical values in it.

- **BMS-Log**: Expressions generated by BMS trained mainly with expressions containing logarithms with constant parameters from the Nguyen dataset.
- **BMS-Trig**: Expressions generated by BMS trained mainly with expressions containing trigonometric functions with constant parameters from the Nguyen dataset.
- **BMS-Sqrt**: Expressions generated by BMS trained mainly with expressions containing square roots with constant parameters from the Nguyen dataset.
- **Nguyen1-8**: The first eight expressions of the Nguyen dataset.

The first observation is that the encode-decode experiment is indeed deterministic. It was run five consecutive times on each of the data sets and the results where completely identical in every aspect. The observations in table 3 a rather underwhelming, showcasing that HVAE struggles to reliably encode and decode the expressions from most datasets. HVAE with NG1-7 performs considerably better in the decoding of expressions which are directly taken from the latent space, however the expressions $sin(x^2) \cdot cos(x) - x/x$ and $\sqrt{x}$ where the only ones recovered from the Nguyen dataset. This does not mean that they are not discoverable with an SR algorithm since other less simplified representations of the expressions might still exist in the latent space.

3.2 Symbolic regression experiments

We move on to the last part of this study, aimed at implementing actual symbolic regression algorithms using HVAE as an expression generator. To recall from section 2.1, we aim to find a function $f : \mathbb{R}^n \to \mathbb{R}$ which fits well to a collection $\mathcal{O} \equiv \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_k, y_k)\}$ of observed data, where $\mathbf{x}_i \in \mathbb{R}^n$ are the independent values and $y_i \in \mathbb{R}$ are the dependent ones. In this section, we focus particularly on data with a single independent variable, i.e. $n = 1$.

We implement two symbolic regression algorithms by extending the random walk model from section 3.1.2. The idea is to add two different heuristics which allow us to accept only certain steps of the walk, thereby directing it to more favorable regions of the latent space. The goal of both algorithms is to minimize the **Mean Squared Error** (MSE) of the eventual output function f with respect to the data in $\mathcal{O}$, which is defined as:

$$\texttt{MSE}_f = \frac{1}{k} \sum_{i=1}^{k} (f(\mathbf{x}_i) - y_i)^2 \tag{3}$$

Note that, for any function f, $\texttt{MSE}_f \geq 0$.

"Accepting" a step t in a random walk refers to enabling the random walk to advance to X_t (3.1.2) only if a certain condition is met. If the step is not accepted, the random walk is forced to return to X_{t-1} and attempt a new step to a different position. The first algorithm (3.2.1) only accepts steps which are "equally good" or "better" than the previous step. The second algorithm (3.2.2) introduces a simulated annealing model [6], which allows to occasionally accept "worse" steps, thereby reducing the probability of converging to a local (and therefore sub-optimal) MSE minimum.

The algorithms will be tested directly on the Nguyen expressions from table 2 and their performance will be compared with the symbolic regression algorithm implemented in [1]. All

experiments in this section will be conducted with HVAE trained with NG1-7.

3.2.1 Restricted random walk

The first version of the SR algorithm adds an MSE based heuristic to the random walk model from 3.1.2. The objective is to find a region within the latent space which decodes into an expression with an MSE as close to 0 as possible. If we denote the expression decoded from the latent space at step t by $Dec(X_t)$, we can modify the random walk equation to:

$$X_t = \begin{cases} X_{t-1} + \delta_t & \text{if} \quad \texttt{MSE}_{Dec(X_{t-1}+\delta_t)} \leq \texttt{MSE}_{Dec(X_{t-1})} \\ X_{t-1} & \text{otherwise} \end{cases} \tag{4}$$

where the MSE is calculated with respect to some input set of observed data $\mathcal{O}$, and δ_t is once again sampled form $N(0, \sigma_{\delta_t}^2 \cdot I_{32})$.

The idea is to let the random walk evolve only if the next proposed step decodes into an expression that's at least as good of a fit to $\mathcal{O}$ as the one obtained from the latest step of the walk. The purpose of having such a restrictive heuristic is to determine wether it is possible to search the latent space without converging to a local MSE minimum at any point.

A local MSE minimum within the latent space is a region which decodes into an expression with an error above 0, but lower than the error associated to any immediately neighboring region. The heuristic implemented in this subsection does not allow for the algorithm to overcome local minimums in the latent space. We can therefore detect local minimums by observing no evolution in the random walk after a large number of iterations.

We add some slight modifications to the implementation of the random walk from section 3.1.2, and extend it by adding the capability of evaluating the Mean Squared Error of a decoded expression. In order to be able to calculate it, we input the observed (or target) data from $\mathcal{O}$ as a 2-row matrix of the form:

$$T = \begin{pmatrix} \mathbf{x}_1 & \mathbf{x}_2 & \cdots & \mathbf{x}_k \\ y_1 & y_2 & \cdots & y_k \end{pmatrix} \quad \forall (\mathbf{x}_i, y_i) \in \mathcal{O}$$

The independent values for the target data are generated via *NumPy*'s `linspace()` method, which creates a set of equally spaced values in some interval $[a, b]$. In our case, a is set to 1, b is set to 2, and we generate 10^3 values. To get the independent values, we either use the Nguyen expressions in table 2, or sample random expressions from the latent space following $N(0, \frac{1}{4} \cdot I_{32})$ and $N(0, I_{32})$. We then evaluate the expressions at each of the previously generated independent values.

The starting point of the walk will be once again set to the origin of the latent space, but the standard deviation σ_{δ_t} for sampling an increment will be modified in order to increase the distance between steps. As we observed in section 3.1.2, it is common to spend multiple steps of a random walk in the same expression region of the latent space. We therefore calculate the MSE only when a change in the decoded expression is detected. For future reference, a step in which the MSE of its corresponding expression is computed will be referred to as an *iteration* of the algorithm. The iteration limit for every run of the algorithm is set to 10^6, and a run is deemed successful if an expression with an MSE of 10^{-15} or less is found.

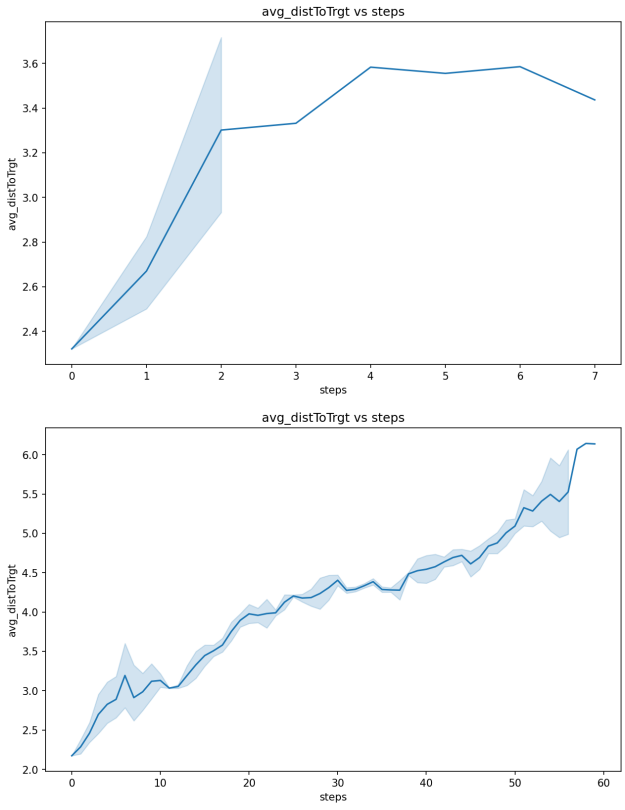
The algorithm is initially tested on 5 expressions sampled from $N(0, \frac{1}{4} \cdot I_{32})$, and another 5 from $N(0, I_{32})$. The step size parameter σ_{δ_t} is fixed to 0.1 and a total of 10 runs are executed for each target expression. We track the number of successful runs, as well as the average number of iterations needed to find a satisfactory expression. Since the targets are directly sampled from the latent space, we can also track how the distance between the random walk and the target evolves.

Table 5. Results of the DRW algorithm run on expressions sampled from the HVAE latent space.

Sampling	**Succesful runs** (out of 50)
$N(0, 0.25 \cdot I_{32})$	48
$N(0, I_{32})$	39

Table 5 shows us that a simple directed random walk algorithm achieves a reasonably good success rate with expressions sampled directly form the latent space. Naturally, the higher the standard deviation, the further away the target is likely to be, and the harder it is to find it.

(a)



(b)

Figure 19. Representation of the average distance with respect to one of the target expressions sampled from (a) $N(0, 0.25 \cdot I_{32})$ and (b) $N(0, I_{32})$.

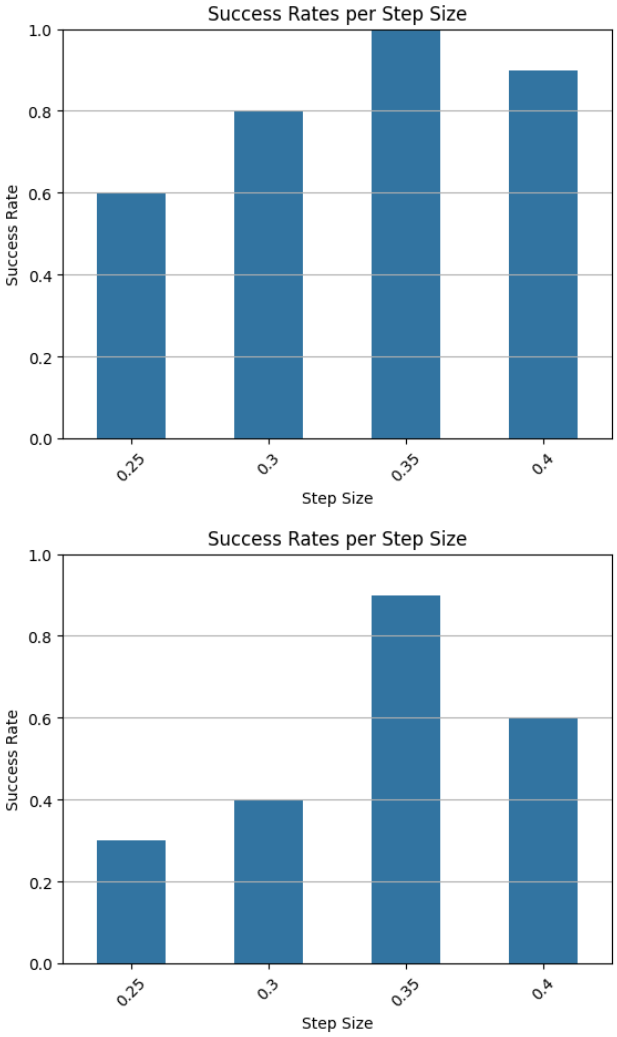
With figure 19 we notice that, even though we achieve good success rates with our SR algorithm,

we don't actually find the exact point of the latent space that we are looking for. In fact, with every accepted step we get further and further away, suggesting that there are a number of redundant regions inside the latent space.

For the Nguyen expressions we once again execute 10 runs per target and count the number of successful ones. The distance-to-target evolution cannot be tracked in this case because the target expressions are not sampled from the latent space, and therefore we cannot know their position. We conduct the experiment with σ_{δ_t} set to 0.25, 0.30, 0.35 and 0.4, and then evaluate how the step size parameter affects the performance of the algorithm.

After running the algorithm on NG-1 to NG-3 for different values of σ_{δ_t}, we determine that $\sigma_{\delta_t} = 0.35$ yields the highest success rate out of all the tested deviations (figure 20).

(a)



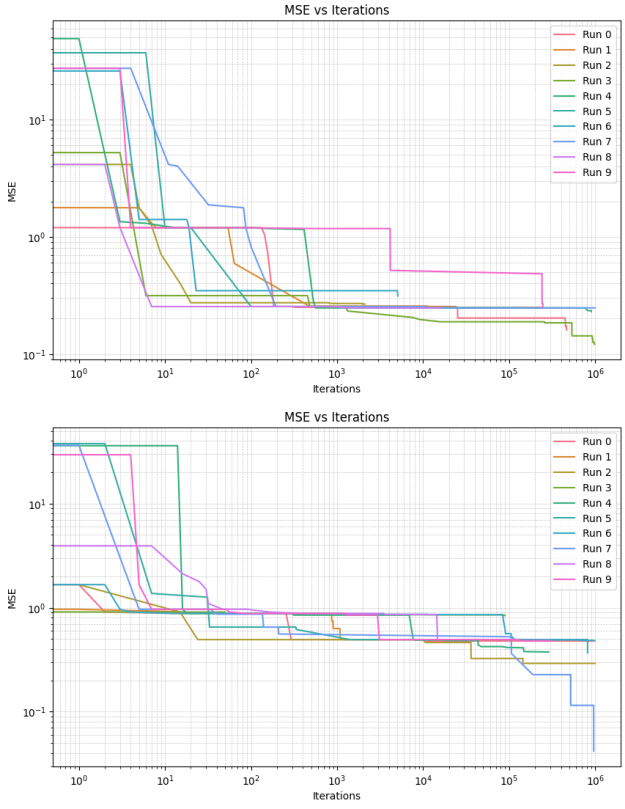
(b)

Figure 20. Comparison of the success rates of the algorithm different values for σ_{δ_t}. (a) DRW ran on NG-2. (b) DRW ran on NG-3.

Table 6. Results of the directed random walk algorithm used on the first 8 expressions of the Nguyen dataset. $\sigma_{\delta_t} = 0.35$

Expression	**Succesful runs** (out of 10)
NG-1	9
NG-2	10
NG-3	9
NG-4	6
NG-5	0
NG-6	0
NG-7	2
NG-8	10

(a)



(b)

Figure 21. The evolution of the MSE during the execution of the DRW algorithm. (a) DRW ran on NG-4. (b) DRW ran on NG-5.

The results in table 20 are similar to ones presented in [1], where a more sophisticated SR algorithm based on genetic programming is implemented. Our main problem is finding expressions NG-5 and NG-6, for which the DRW algorithm gets repeatedly stuck in local MSE minima. The long

horizontal lines in figure 21 indicate that the algorithm accepts solutions which do not improve the MSE for large portions of time. This further exposes the expression redundancy problem that HVAE has.

3.2.2 Simulated annealing

For the final experiment carried out in this study, we extend the previously shown SR algorithm by implementing a simulated annealing model for accepting new steps during the random walk. Simulated annealing (SA) [6] is a probabilistic model aimed at finding approximately optimal values of a function given a large search domain. It specifically implements a way of overcoming (or skipping) local minima/maxima that may appear in the search space. The technique gets its name due to the way it simulates a system gradually ”cooling down” over time, which results in changes to its physical properties. In the context of search algorithms, the ”cooling down” refers to the gradual decrease in the probability of accepting solutions worse than the most current one.

Simulated annealing is governed by a probability equation which determines how likely it is for a proposed solution to be accepted. We can define a cost function E that can evaluate the quality of any given solution to our problem. Our goal is to find a solution with the lowest possible cost assigned to it. Suppose we have a candidate solution s, and a new candidate s' is proposed. SA defines the probability of accepting s' as a new solution with the equation

$$P_a(E, s, s') = \begin{cases} e^{-\frac{E(s')-E(s)}{T}} & \text{if} \quad E(s') > E(s) \\ 1 & \text{otherwise} \end{cases} \tag{5}$$

where $E(s)$ is the cost of the solution s and $T > 0$ is the temperature of the system. Notice that, if s' is better than s (lower cost), it is automatically accepted. However, s' can still be accepted if it's a worse solution than s. Occasionally accepting worse solutions enhances the exploration of the solution space of a problem since local optima become surmountable. This probabilistic model is used in the famous Metropolis-Hastings algorithm [7].

The acceptance probability function is well defined, since

$$T > 0,\ E(s') > E(s) \implies -\frac{E(s') - E(s)}{T} < 0 \implies 0 < e^{-\frac{E(s')-E(s)}{T}} < 1$$

The higher the value of $E(s') - E(s)$ is, the worse the solution s' is relative to s, thereby the lower the probability of acceptance is. The temperature parameter T is gradually reduced closer to 0 as the search progresses, which results in slowly restricting the acceptance probability for worse solutions. This process is how the simulation of a system cooling down to some stable (optimal) state is implemented. In practice, the temperature parameter is reduced according to some specified *cooling function*.

The cost function E in our case is the MSE of an expression decoded from the latent space. As we saw in section 3.2.1, it is likely to find sub-optimal MSE minima inside the latent space. Adding a simulated annealing model to the restricted random walk algorithm from section 3.2.1 might help in overcoming said minima and therefore achieving better results.

We reuse the implementation of the SR algorithm from 3.2.1 and include a simple SA model to it. Instead of immediately rejecting a step $X_{t-1} + \delta_t$ with a worse expression than X_{t-1}, we first calculate

the difference between the MSEs:

$$\Delta\texttt{MSE} = \texttt{MSE}_{Dec(X_{t-1}+\delta_t)} - \texttt{MSE}_{Dec(X_{t-1})}$$

We then use *NumPy* to calculate the acceptance probability $P_a(X_{t-1}+\delta_t)$ and to uniformly generate a random number between 0 and 1, denoted by $\texttt{rand}(0,1)$. The acceptance probability is computed as

$$P_a(X_{t-1}+\delta_t) = \begin{cases} e^{-\frac{\Delta\texttt{MSE}}{T}} & \text{if} \quad \texttt{MSE}_{Dec(X_{t-1}+\delta_t)} > \texttt{MSE}_{Dec(X_{t-1})} \\ 1 & \text{otherwise} \end{cases}$$

We accept the new step if $\texttt{rand}(0,1) \leq P_{X_t}(X_{t-1})$. The evolution of the algorithm can therefore be written as

$$X_t = \begin{cases} X_{t-1}+\delta_t & \text{if} \quad \texttt{rand}(0,1) \leq P(X_{t-1}+\delta_t) \\ X_{t-1} & \text{otherwise} \end{cases}$$

Since we can now accept ”bad” changes in the expressions, we keep track of the best found expression during the entire walk and update it accordingly. The temperature T is cooled down according to the cooling function $T_s = r \cdot T_{s-1}$, where T_s is the temperature at stage s and $r \in (0,1)$ is some fixed cooling rate.

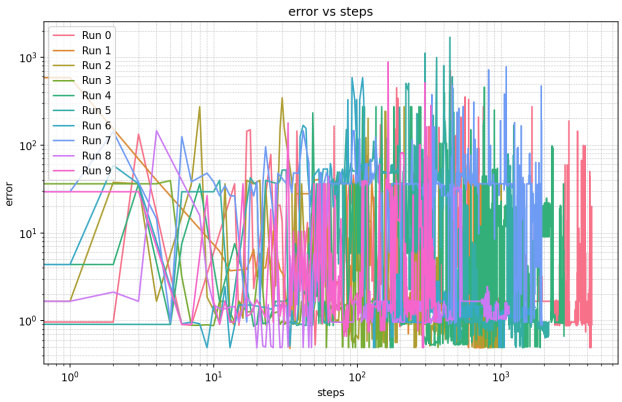
We test this algorithm on the 8 Nguyen expressions from table 2 and once again set the iteration limit to 10^6. A cooling stage will be applied once every 10^3 iterations, giving us a total of 10^3 stages. Since we are more likely to accept changes, we are more likely to travel further into the latent space. We do not want our distance from the origin to increase too quickly, because that would take us to the regions of the latent space where expressions are more complex without having explored the center of the space properly. In section 3.2.1, larger steps proved to be more effective, but the possible divergence caused by them was compensated with a very restrictive heuristic that didn’t allow for much free movement. This will not be the case for the SA algorithm, hence we limit the value of σ_{δ_t} to 0.1.

Our main focus are the expressions NG-5, NG-6 and NG-7 since the rest of the Nguyen expressions can be recovered with a high success rate using the DRW algorithm. Figure 22 clearly shows that a temperature of $T = 1000$ is way too high since bad solutions are constantly accepted and the algorithm becomes a random walk.

Table 7 shows the results with a highly reduced initial temperature being used. The success rate of finding NG-4 is significantly reduced, but the one for NG-7 is increased. Other than that, no significant changes in the performance are observed.

The algorithm was run with a wide range of values for the SA parameters, but no greater results were observed. We can therefore conclude that HVAE is not able to effectively generate all expressions of the Nguyen dataset. We deem the results of the symbolic regression problem applied to the Nguyen benchmark as not completely satisfactory, but future research and improvements on the model could most definitely yield better results.

(a)



(b)

Figure 22. The evolution of the MSE during the runs of SA algorithm with $T = 10^3$ and $r = 0.975$ (a) DRW ran on NG-4. (b) DRW ran on NG-5.

Table 7. Results of the simulated annealing algorithm used on the first 8 expressions of the Nguyen dataset. $T = 0.05$, $r = 0.975$.

Expression	**Succesful runs** (out of 10)
NG-1	10
NG-2	9
NG-3	8
NG-4	1
NG-5	0
NG-6	0
NG-7	6
NG-8	10

4 Conclusions

In this project, we set out to investigate the performance and characteristics of Hierarchical Variational Autoencoder (HVAE) when applied to the generation and representation of mathematical expressions. Through a series of experiments—including latent space sampling, random walks, and symbolic regression algorithms—we examined how effectively can HVAE capture and generalize over the symbolic structure of mathematical domains.

Our findings show that HVAE is indeed capable of learning meaningful latent representations that reflect some aspects of mathematical expression structure. By sampling points from the latent space and decoding them into expressions, we observed coherent trends such as increasing expression complexity with higher sampling variance and local consistency in regions of low complexity. Additionally, the random walk experiments demonstrated HVAE's potential for exploring symbolic space incrementally, uncovering both familiar and novel expressions along the way.

However, the results are not entirely satisfying. A major limitation revealed by our analysis is the inconsistency and irregularity in the latent spaces learned by the model. While some regions of the latent space produce smoothly varying and meaningful expressions, others show undesirable properties such as redundancy, sharp discontinuities, or excessive concentration of similar expressions. In particular, our frequency and complexity analysis, as well as the symbolic regression algorithms that were implemented, showed that the model often re-generates the same expressions repeatedly, and that certain expressions dominate the latent space, potentially due to lackluster training or limitations in the HVAE architecture itself.

These inconsistencies suggest that although the HVAE framework holds promise for symbolic reasoning tasks, further refinement is necessary to make it reliable and effective for applications such as symbolic regression. Future work could explore improved training objectives, architectural modifications, or more sophisticated approaches that combine HVAE with symbolic priors or grammar-based constraints to mitigate the weaknesses identified in this study.

Overall, this project provides a solid foundation for understanding how HVAE behaves in the symbolic domain and highlights both its potential and its current limitations. As for the objectives set out at the start of this study, we can confidently state that they have been successfully met during the development of this project.

Words of appreciation go once more to the supervisors of this project, Dr. Roger Guimerà Manrique and Dr. Marta Sales Pardo, whose continuous guidance and availability have been of most significance for the successful development of this study.

References

[1] S. Mežnar, S. Džeroski, and L. Todorovski, "Efficient generator of mathematical expressions for symbolic regression," *Machine Learning*, vol. 112, no. 11, pp. 4563–4596, 2023, ISSN: 1573-0565. DOI: 10.1007/s10994-023-06400-2. [Online]. Available: https://doi.org/10.1007/s10994-023-06400-2.

[2] Wikipedia contributors, *Symbolic regression — Wikipedia, the free encyclopedia*, https://en.wikipedia.org/w/index.php?title=Symbolic_regression&oldid=1286038942, 2025.

[3] R. Guimerà, I. Reichardt, A. Aguilar-Mogas, *et al.*, "A bayesian machine scientist to aid in the solution of challenging scientific problems," *Science Advances*, vol. 6, no. 5, eaav6971, 2020. DOI: 10.1126/sciadv.aav6971. eprint: https://www.science.org/doi/pdf/10.1126/sciadv.aav6971. [Online]. Available: https://www.science.org/doi/abs/10.1126/sciadv.aav6971.

[4] M. Virgolin and S. P. Pissis, "Symbolic regression is NP-hard," *Transactions on Machine Learning Research*, 2022, ISSN: 2835-8856. [Online]. Available: https://openreview.net/forum?id=LTiaPxqe2e.

[5] N. Uy, N. Hoai, M. O'Neill, R. McKay, and E. Galván-López, "Semantically-based crossover in genetic programming: Application to real-valued symbolic regression," English, *Genetic Programming and Evolvable Machines*, vol. 12, no. 2, pp. 91–119, Jun. 2011, ISSN: 1389-2576. DOI: 10.1007/s10710-010-9121-2.

[6] Wikipedia contributors, *Simulated annealing — Wikipedia, the free encyclopedia*, https://en.wikipedia.org/w/index.php?title=Simulated_annealing&oldid=1292887544, 2025.

[7] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, "Equation of state calculations by fast computing machines," *The journal of chemical physics*, vol. 21, no. 6, pp. 1087–1092, 1953.

Appendices

A Code Availability

All the code used for this study, including instructions on how to run it, is available in the following repository: GitHub Repository