Leonel Fernando Nabaza Ruibal

Approximating distributions of closed-form mathematical expressions for symbolic regression

Bachelor's Degree Thesis

Directed by: Dr. Roger Guimerà Manrique

Dr. Marta Sales Pardo

Bachelor's Degree in Mathematical and Physical Engineering

UNIVERSITAT ROVIRA i VIRGILI

Tarragona

June 6, 2025

“Imagination is more important than knowledge. For knowledge is limited, whereas imagination embraces the entire world, stimulating progress, giving birth to evolution.”

— Albert Einstein

Acknowledgements

I would like to thank my supervisors, Dr. Marta Sales Pardo and Dr. Roger Guimerà Manrique, for their implication, guidelines and for being very supportive and approachable during all the process. It was thanks to them that I got so far.

Resum

La regressió simbòlica és una eina potent per descobrir models analítics a partir de dades sense necessitat de conèixer prèviament el sistema subjacent. Aquest treball explora un enfocament innovador per millorar l'eficiència del mostreig d'expressions matemàtiques en forma tancada dins del marc de la regressió simbòlica. A partir del *Bayesian Machine Scientist* (BMS) —un mètode probabilístic per seleccionar models basat en la longitud de descripció— es proposa i implementa una nova estructura probabilística anomenada *probability tree*. Aquesta estructura genera expressions matemàtiques mitjançant el mostreig recursiu d'operacions, variables i paràmetres segons distribucions de probabilitat específiques de cada node. L'arbre de probabilitat s'entrena amb models generats tant pel BMS com per ell mateix, mitjançant la minimització de la divergència de Kullback–Leibler entre les longituds de descripció assignades pel BMS i les probabilitats assignades per l'arbre. Els resultats mostren que el mètode és capaç d'aprendre eficaçment a partir dels models proporcionats. Com a resultat, permet una exploració més eficient i estructurada de l'espai de models. Aquests resultats posen de manifest el potencial dels arbres de probabilitat com a alternativa o complement als enfocaments tradicionals de regressió simbòlica.

Paraules clau: Aprenentatge automàtic, Regressió simbòlica, Sistemes complexos

Abstract

Symbolic regression is a powerful tool for uncovering analytical models from data without prior knowledge of the underlying system. This thesis explores a novel approach to improve the efficiency of sampling closed-form mathematical expressions within the symbolic regression framework. Building upon the Bayesian Machine Scientist (BMS) — a probabilistic method for selecting models based on description length — a new probabilistic structure called the *probability tree* is proposed and implemented. This structure generates mathematical expressions by recursively sampling operations, variables, and parameters based on node-specific probability distributions. We train the probability tree using models generated by both the BMS and itself, by minimizing the Kullback–Leibler divergence between the BMS-assigned description lengths and the tree-assigned probabilities. Results show that the method is able to effectively learn from the training models. As a result, it enables a more efficient and structured exploration of the model space. These outcomes highlight the potential of probability trees as an alternative or complement to traditional symbolic regression techniques.

Keywords: Machine Learning, Symbolic Regression, Complex Systems

Resumen

La regresión simbólica es una herramienta poderosa para descubrir modelos analíticos a partir de datos sin necesidad de conocer previamente el sistema subyacente. Este trabajo explora un enfoque innovador para mejorar la eficiencia del muestreo de expresiones matemáticas en forma cerrada dentro del marco de la regresión simbólica. A partir del *Bayesian Machine Scientist* (BMS) —un método probabilístico para seleccionar modelos basado en la longitud de descripción— se propone e implementa una nueva estructura probabilística denominada *probability tree*. Esta estructura genera expresiones matemáticas mediante el muestreo recursivo de operaciones, variables y parámetros según distribuciones de probabilidad específicas de cada nodo. El árbol de probabilidad se entrena con modelos generados tanto por el BMS como por él mismo, mediante la minimización de la divergencia de Kullback–Leibler entre las longitudes de descripción asignadas por el BMS y las probabilidades asignadas por el árbol. Los resultados muestran que el método es capaz de aprender eficazmente a partir de los modelos proporcionados. Como resultado, permite una exploración más eficiente y estructurada del espacio de modelos. Estos resultados ponen de manifiesto el potencial de los árboles de probabilidad como una alternativa o complemento a los enfoques tradicionales de regresión simbólica.

Palabras clave: Aprendizaje automático, Regresión Simbólica, Sistemas Complejos

Contents

Resum iii

Abstract iv

Resumen v

Contents vi

1 Introduction and Theoretical Background 2

1.1 Symbolic Regression . . . 3

1.1.1 Trees for Mathematical Expressions . . . 3

1.2 Bayesian Machine Scientist . . . 6

1.3 Objectives . . . 13

2 Results 15

2.1 Theoretical Results . . . 15

2.1.1 Probability Trees . . . 15

2.2 Experimental Results . . . 20

2.2.1 Use of the Bayesian Machine Scientist . . . 20

2.2.2 Datasets and Data Generation . . . 26

2.2.3 Probability Tree Implementation . . . 30

2.2.4 Probability Tree Training and Testing . . . 34

2.2.5 Analysis of the Training . . . 40

3 Conclusions 43

3.1 Ethics of the Project . . . 45

References 45

A Appendix **48**
A Probability Tree Implementation 48
B Alternative Strategy for the Probability Tree Training 48

Chapter 1

Introduction and Theoretical Background

Symbolic regression is a rapidly growing subfield within machine learning (ML). It navigates through the space of mathematical expressions to find a closed-form model that best describes a given dataset. In order to find this model, its accuracy and simplicity are taken into account ([21]).

One of the most remarkable characteristics of symbolic regression is that it combines techniques and processes from diverse scientific fields to produce an analytical expression that best describes the data without the need to incorporate prior knowledge about the investigated system. Thus, it can "deduce" or discover profound relations between variables that can be generalizable, applicable, explainable and span over most scientific, technological, economical and social principles [1].

As for the general problem that symbolic regression deals with, there have been many different approaches proposed. Nevertheless, there is one common challenge that all these approaches face: the strategy by which the model space is navigated and how this exploration should change in the process.

In this thesis, based on the use of the Bayesian Machine Scientist, [6] a new approach for the exploration of the sample space is proposed and studied.

This new approach, which will be further developed in the following sections, consists of the implementation of a tree-like structure where each node is assigned a different probability for choosing an operator, parameter or variable, and these probabilities are adjusted via the minimization of a Kullback–Leibler divergence ([18]). Expressions then are built by walking through the tree and choosing operators according to the probability of each node.

1.1 Symbolic Regression

As previously mentioned, symbolic regression aims to find the closed-form mathematical expression that best fits a given dataset, uncovering underlying relationships in the form of human-interpretable equations.

Definition 1.1 (The symbolic regression problem)**.** Let $\mathcal{D} = \{(x^{(i)}, y^{(i)})\}_{i=1}^{n} \subset \mathbb{R}^k \times \mathbb{R}$ be a dataset where each $x^{(i)} = (x_1^{(i)}, \ldots, x_k^{(i)})$ is an input vector and $y^{(i)}$ is its corresponding output. This dataset describes an unknown property $y = F(x, \theta)$, where x is the set of k variables and $\theta \in \mathbb{R}^l$ is the set of parameters. Let $\mathcal{F} = \{+, -, \times, \div, \sin, \cos, \exp, \ldots\}$ be a set of operators (the term includes functions, variables, constants and operations). Our objective is to find a mathematical expression $f^* \in \mathcal{E}(\mathcal{F})$, where $\mathcal{E}(\mathcal{F})$ is the space of all expressions that can be generated with the set $\mathcal{F}$, that minimizes an error function $\mathcal{L}(f, \mathcal{D}) = \dfrac{1}{n} \displaystyle\sum_{i=1}^{n} \ell\left(f(x^{(i)}), y^{(i)}\right)$.

Over history, scientists have found such expressions manually, by observing the patterns in the data, proposing a model and testing its accuracy (for example Newton's second law: $F = m \cdot a$). With the improvement of computational capacities in recent decades, symbolic regression has shifted towards an approach in which a vast space of analytic expressions is explored. This idea was firstly explored during the 1970s and 1980s, in projects such as [4] and [9]. In the beginning, approaches based on brute force and heuristic strategies were used and provided mathematical equations with good results for simple physical laws. Later, genetic algorithms were employed, expanding the size of the model space to be explored. In the following subsections, I will introduce key components to tackle this problem.

1.1.1 Trees for Mathematical Expressions

In order to break down mathematical expressions into chunks that can be analysed and used algorithmically, for making them evolve according to how the error changes, a common approach is to represent them as binary trees.

Definition 1.2 (Tree)**.** In graph theory, a *tree* is an undirected graph in which any pair of vertices is connected via a unique path: a connected acyclic undirected graph.

Definition 1.3 (Rooted tree)**.** A tree is a *rooted tree* if one of its vertices has been designated as the root, and if in a rooted three a vertex v is the vertex that immediately precedes w on the path from the root to w, then v is the parent of w and w is a child of v.

Definition 1.4 (k-ary tree)**.** A *k-ary* tree is a rooted tree in which each internal vertex has at most k children, and in which at least one vertex has exactly k children. A 2-ary tree is usually called a binary tree.

A systematic visit to the vertices of a tree is called a *tree traversal*. In the case of binary trees, there are three types of traversals: *preorder*, *inorder* and *postorder*. The three traversals are distinguished basically in the way they explore the two children of each vertex. For their explanation, the following terminology is used [16]: T is the binary tree of root r, T_1 and T_2 are the subtrees of roots v_1 and v_2 induced by the children of root r, where v_1 is on the left and v_2 on the right in a plane drawing of the tree. These traversals are defined recursively in the following way.

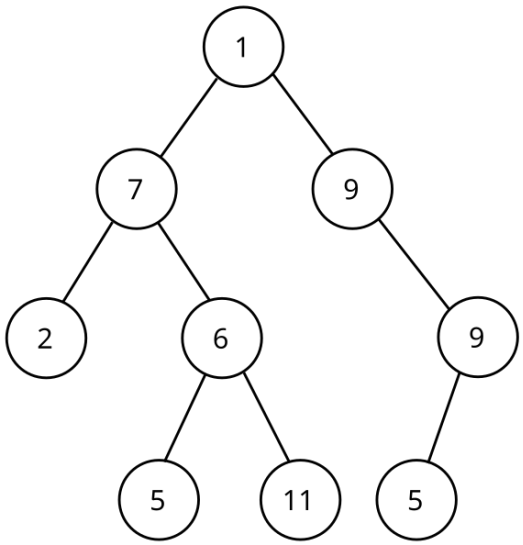


Figure 1.1: Binary tree example

- Preorder traversal of T:
  1. List the root r.
  2. Perform a preorder traversal of the left subtree T_1.
  3. Perform a preorder traversal of the right subtree T_2.
- Inorder traversal of T:
  1. Perform an inorder traversal of the left subtree T_1.
  2. List the root r.

3. Perform an inorder traversal of the right subtree T_2.

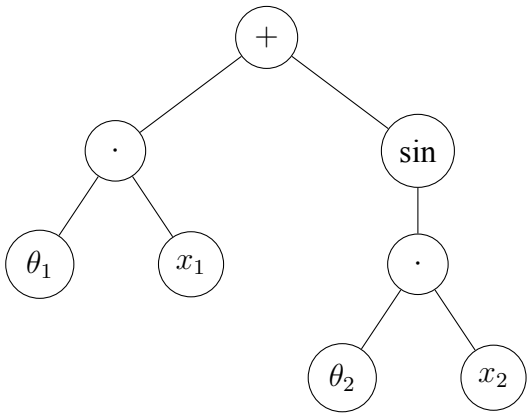
- Postorder traversal of T:

1. Perform a postorder traversal of the left subtree T_1.
2. Perform a postorder traversal of the right subtree T_2.
3. List the root r.

Example 1.1. Reading the tree in the figure 1.1 in these gives arouses the following results:

- Preorder: 1, 7, 2, 6, 5, 11, 9, 9, 5.
- Inorder: 2, 7, 5, 6, 11, 1, 9, 9, 5.
- Postorder: 2, 5, 11, 6, 7, 5, 9, 9, 1.

Example 1.2. The function $f(x_1, x_2) = \theta_1 \cdot x_1 + \sin(\theta_2 \cdot x_2)$ can be expressed using inorder as



Working with this structure offers several advantages. First of all, there is a natural flexibility to represent complex mathematical expressions, as binary trees can capture both binary operators (like $+$ or $\cdot$) and unary functions (like sin, exp). Additionally, the recursive nature of trees allows for straightforward evaluation of expressions by traversing from leaves to root. In the context of symbolic regression and genetic programming, binary trees make it easy to apply structural modifications, such as mutation and crossover, by replacing or swapping entire subtrees. This enables exploration of the search space. Moreover, trees provide a clear measure of expression complexity through metrics like depth or node count, which can be used to control overfitting. Finally, subtrees can be evaluated or simplified independently, allowing for parallelization and optimization of computation.

1.2 Bayesian Machine Scientist

Definition 1.5 (Machine Scientist)**.** A *machine scientist* refers to an automated system or algorithm that can independently discover, formulate, and test scientific hypotheses or equations directly from raw data, much like a human scientist would.

Despite the remarkable progress that has been achieved in the last decades, algorithms for symbolic regression have two main problems. First of all, they have to equilibrate both the fit and the complexity of the proposed model and avoid overfitting as well. Second, according to [6], *machine scientists*, the tool used to perform symbolic regressions, should explore a considerably large space of closed-form mathematical models.
In the previously referenced article, [6], the authors propose a Bayesian approach to these problems: a *Bayesian Machine Scientist* (BMS). To tackle the trade-off between goodness of fit and model complexity, they compute the posterior probability of each candidate expression by combining basic probabilistic principles with explicit approximations. This posterior naturally balances both the fit to the data and a prior distribution over mathematical expressions that penalizes overly complex models. To construct this prior, they compile a large corpus of closed-form expressions from Wikipedia and apply a maximum entropy principle to ensure statistical consistency with the observed distribution of expressions [7]. Additionally, to efficiently explore the vast space of possible equations, they introduce a Markov Chain Monte Carlo (MCMC) algorithm that samples expressions according to their posterior probabilities, allowing the discovery of interpretable and data-consistent models. This approach has managed to obtain the true generating model when fed with synthetic data, surpassing the efficacy of other modern machine scientists. Moreover, it also performs better for uncovering closed-form mathematical models for which no previous model has been agreed on and is able to provide more accurate predictions for values out of the sample [14].
As for the formulation of the problem for the BMS, it follows a similar idea as in 1.1, but we assume that the experimental values for the dataset have some experimental error $y^{(i)} = F(x^{(i)}, \theta) + \varepsilon^{(i)}$. Before proceeding further I will introduce some concepts.

Definition 1.6 (Marginalization)**.** Given a joint distribution of two random variables, X and Y, the marginal distribution of either variable, say X, is the probability distribution of X when the values of Y are not taken into consideration. If the random variables are discrete that is

$$p(X = x_i) = \sum_{y \in Y} p(x_i, y).$$

If the random variables are continuous, the probability density function can be written as

$$p(x) = \int_{y \in \text{Range}(Y)} p(x, y) \, \mathrm{d}y$$

Definition 1.7 (Chain rule of probability for two random variables)**.** Let A and B be two random variables. The *chain rule of probability* expresses the joint probability $P(A, B)$ as the product of a conditional probability and a marginal probability.

$$P(A, B) = P(A|B) \; P(B),$$

or equivalently

$$P(A, B) = P(B|A) \; P(A).$$

Definition 1.8 (Bayes' rule)**.** Baye's rule is a fundamental principle in probability theory that describes how to update the probability of a hypothesis or event based on new evidence or information. It provides an expression to compute the *conditional probability* -the probability of an event occurring given that another event has already occurred-. The expression is:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}. \tag{1.1}$$

In the context in which A represents a model and B the data, each term has a specific name:

- $P(A|B)$ is the *posterior probability*: the probability of event A given that B has occurred.
- $P(B|A)$ is the *likelihood*: the probability of event B given that A is true.
- $P(A)$ is the *prior probability* of event A before considering event B.
- $P(B)$ is the *marginal probability* or *evidence*: the total probability of event B occurring.

Proposition 1.1. The BMS assigns a probability to every possible closed-form mathematical expression f_i a *plausibility33* $p(f_i|D)$ given by the marginal posterior

$$p(f_i|D) = \frac{1}{Z} \int_{\Theta_i} \mathrm{d}\theta_i \; p(D|f_i, \theta_i) \; p(\theta_i|f_i) \; p(f_i) = \frac{\exp\left(-\mathcal{L}(f_i)\right)}{Z}, \tag{1.2}$$

where θ_i are the parameters associated with expression f_i, the integral is over the space Θ_i

of possible values of these parameters, $Z = P(D)$ and

$$\mathcal{L}(f_i) \equiv -\log\left(p(D, f_i)\right) = -\log\left(\int_{\Theta_i} \mathrm{d}\theta_i \; p(D|f_i, \theta_i)\; p(\theta_i|f_i)\; p(f_i)\right) \tag{1.3}$$

Proof. We first note that we can express $p(f_i|D)$ as the marginalization of $p(f_i, \theta_i|D)$ with respect to the parameters and then apply Bayes' Rule.

$$\begin{aligned}
p(f_i|D) &= \int_{\Theta_i} \mathrm{d}\theta_i \; p(f_i, \theta_i|D) \\
&= \int_{\Theta_i} \mathrm{d}\theta_i \; \frac{p(D|f_i, \theta_i)\; p(f_i, \theta_i)}{p(D)} \\
&= \frac{1}{p(D)} \int_{\Theta_i} \mathrm{d}\theta_i \; p(D|f_i, \theta_i)\; p(f_i, \theta_i) \\
&= \frac{1}{p(D)} \int_{\Theta_i} \mathrm{d}\theta_i \; p(D|f_i, \theta_i)\; p(\theta_i|f_i)\; p(f_i) \\
&= \frac{1}{p(D)} \exp\left(\log\left(\int_{\Theta_i} \mathrm{d}\theta_i \; p(D|f_i, \theta_i)\; p(\theta_i|f_i)\; p(f_i)\right)\right) \\
&= \frac{1}{p(D)} \exp\left(-(-1)\log\left(\int_{\Theta_i} \mathrm{d}\theta_i \; p(D|f_i, \theta_i)\; p(\theta_i|f_i)\; p(f_i)\right)\right) \\
&= \frac{\exp\left(-\mathcal{L}(f_i)\right)}{Z}.
\end{aligned}$$

□

$\mathcal{L}(f_i)$ is know as the *description length* of the model f_i, and is the number of nats[1] needed to jointly encode the data and the model with an optimal code. It is important to understand that the description length cannot be calculated exactly. First of all, because we are integrating over all the parameters and this integral does not always have a closed form. Secondly, the function space is combinatorially vast and heterogeneous to explore. So, approximations are used to round this value. In the case of the BMS, the chosen approximation is

$$\mathcal{L}(f_i) \approx \frac{B(f_i)}{2} - \log\left(p(f_i)\right), \tag{1.4}$$

where

[1] The term *nat* comes from *natural unit of information* and it is a unit of information or information entropy based on natural logarithms and powers of e [20].

Definition 1.9 (Bayesian Information Criterion, BIC)**.**

$$B(f_i) = q \log(n) - 2 \log\left(\hat{L}\right) \tag{1.5}$$

and

- $\hat{L}$ is the maximized value of the likelihood function.
- n is the number of data points in the dataset.
- q is the number of parameters.

The approximation 1.4 has been obtained by considering that the likelihood $p(D|f_i, \theta_i)$ is sharply peaked around the maximum likelihood parameters θ_i^* and that the prior $p(\theta_i|f_i)$ varies slowly in that region. Under these conditions, the integral can be approximated by expanding the logarithm of the likelihood in a Taylor series around θ_i^* up to second order and performing a Gaussian integral. Let us see it:

$$\log\left(p(D|f_i, \theta_i)\right) \approx \log\left(\hat{L}\right) - \frac{n}{2}(\theta_i - \theta_i^*)^\top H_{\log(p(D|f_i,\theta_i))}|_{\theta_i=\theta_i^*}(\theta_i - \theta_i^*)$$

Where H stands for the hessian matrix. It can also be written as $\mathcal{I}(\theta_i^*)$, known as the *observed information* [17]. There is not first order term since we are evaluating at the maximum then the first derivative term is zero. Now,

$$\begin{aligned}
\mathcal{L}(f_i) &= -\log\left(p(D, f_i)\right) = -\log\left(\int_{\Theta_i} \mathrm{d}\theta_i \; p(D|f_i, \theta_i)\; p(\theta_i|f_i)\; p(f_i)\right) \\
&= -\log\left(p(f_i)\int_{\Theta_i} \mathrm{d}\theta_i \; p(D|f_i, \theta_i)\; p(\theta_i|f_i)\right) \\
&= -\log\left(p(f_i)\int_{\Theta_i} \mathrm{d}\theta_i \; \exp\left(\log\left(p(D|f_i, \theta_i)\; p(\theta_i|f_i)\right)\right)\right) \\
&\approx -\log\left(p(f_i)\int_{\Theta_i} \mathrm{d}\theta_i \; \exp\left(\log\left(\hat{L}\right) - \frac{n}{2}(\theta_i - \theta_i^*)^\top H_{\log(p(D|f_i,\theta_i))}|_{\theta_i=\theta_i^*}(\theta_i - \theta_i^*)\right) p(\theta_i|f_i)\right) \\
&\approx -\log\left(p(f_i)p(\theta_i^*|f_i)\hat{L}\left(\frac{2\pi}{n}\right)^{q/2}\left|H_{\log(p(D|f_i,\theta_i))}|_{\theta_i=\theta_i^*}\right|^{-1/2}\right) \\
&= -\log\left(p(f_i)\right) - \left(\log\left(p(\theta_i^*|f_i)\right) + \log\left(\hat{L}\right) + \frac{q}{2}\log(2\pi) - \frac{q}{2}\log(n) - \frac{1}{2}\log\left(\left|H_{\log(p(D|f_i,\theta_i))}|_{\theta_i=\theta_i^*}\right|\right)\right.
\end{aligned}$$

where we have considered that $p(\theta_i|f_i)$ is flat around its maximum. As n increases, the

constant terms $\log(2\pi)$, $\log\left(\left|H_{\log(p(D|f_i,\theta_i))}|_{\theta_i=\theta_i^*}\right|\right)$ become less relevant, so

$$\begin{aligned}\mathcal{L}(f_i) &\approx -\log(p(f_i)) - \left(\log\left(\hat{L}\right) - \frac{q}{2}\log(n)\right) \\ &= -\log(p(f_i)) + \frac{1}{2}\left(q\log(n) - 2\log\left(\hat{L}\right)\right) \\ &= -\log(p(f_i)) + \frac{B(f_i)}{2} \\ &= \frac{B(f_i)}{2} - \log(p(f_i))\end{aligned}$$

In practice, the BIC is computed assuming that the points in the dataset that come from an experimental measure carry a Gaussian distributed error (with mean 0 and variance σ^2). In order to do that it is important to find the variance that allows us to obtain the greatest likelihood ($p(D|f_i,\theta_i^*) = \hat{L}$), which is computed by:

$$\begin{aligned}p(D|f_i,\theta_i) &= \prod_{j=1}^{n}\frac{1}{\sqrt{2\pi\sigma^2}}\exp\left(-\frac{(y^{(j)} - f_i(x^{(j)},\theta_i)^2}{2\sigma^2}\right) \\ &= \frac{1}{(\sqrt{2\pi\sigma^2})^n}\exp\left(-\frac{\sum_{j=1}^{n}(y^{(j)} - f_i(x^{(j)},\theta_i)^2}{2\sigma^2}\right)\end{aligned}$$

$$\begin{aligned}\frac{\partial p(D|f_i,\theta_i)}{\partial\sigma} &= \frac{1}{(\sqrt{2\pi})^n}\exp\left(-\frac{\sum_{j=1}^{n}(y^{(j)} - f_i(x^{(j)},\theta_i)^2}{2\sigma^2}\right) \\ &\quad\cdot\left(\frac{-\frac{\sum_{j=1}^{n}(y^{(j)} - f_i(x^{(j)},\theta_i)^2}{2}\cdot(-2)\sigma^{-3}\sigma^n - n\sigma^{n-1}}{\sigma^{2n}}\right) = 0 \\ &\to -\frac{\sum_{j=1}^{n}(y^{(j)} - f_i(x^{(j)},\theta_i)^2}{2}\cdot(-2)\sigma^{-3}\sigma^n - n\sigma^{n-1} = 0 \\ &= \sum_{j=1}^{n}(y^{(j)} - f_i(x^{(j)},\theta_i)^2\cdot\sigma^{n-3} - N\sigma^{n-1} = 0 \\ &\to \sigma^2 = \frac{\sum_{j=1}^{n}(y^{(j)} - f_i(x^{(j)},\theta_i)^2}{n}\end{aligned}$$

Plugging this value of the variance into the BIC expression 1.5 gives us

$$B(f_i) = q\log(n) + n\left(1 + \log\left(\frac{\sum_{j=1}^{n}(y^{(j)} - f_i(x^{(j)},\theta_i)^2}{n}\right) + \log(2\pi)\right) \tag{1.6}$$

Remark 1.1. It is important to take into account that the parameters of the complete model are

$\theta_1, \ldots, \theta_l$ and σ, because its value is also unknown. So in this case we would have $q = l + 1$.

In order to achieve the maximum possible likelihood, the parameters are found via least squares method.

Remark 1.2. The objective is to find the best possible function given some data, so we aim to find the function f_i with the highest possible $p(f_i|D)$, which is proportional to $e^{-\mathcal{L}(f_i)}$, $p(f_i|D) \propto e^{-\mathcal{L}(f_i)}$. This maximum is achieved when the description length is minimal (dividing by $p(D)$ represents to apply a normalization factor). This is analogous to finding a minimum energy configuration in statistical mechanics, where the most probable state corresponds to the one minimizing the energy function (H). In fact, if we wanted to compute the value of $p(D)$, we could take into account this analogy and compute it as the *partition function* of the system:

Symbolic Regression Concept	**Statistical Mechanics Connection**
$p(f_i\|D) = \dfrac{e^{-\mathcal{L}(f_i)}}{p(D)} \propto e^{-\mathcal{L}(f_i)}$	$p(s) = \dfrac{e^{-\beta H(s)}}{Z} \propto e^{-\beta H(s)}$
$p(D) = \sum_i e^{-\mathcal{L}(f_i)}$	$Z = \sum_s e^{-\beta H(s)}$

Table 1.1: Correspondence between symbolic regression and statistical mechanics.

Nevertheless, because of the point made in the remark, in the present context, it is not necessary to compute $p(D)$.

As for the expression sampling method, a Metropolis-Hastings algorithm [19], which is based in the Markov Chain Monte Carlo method was implemented in order to explore the model space in a representative way. This algorithm allows the BMS to sample expressions proportionally to their posterior plausibility given the data, thus balancing model accuracy and complexity.
As described earlier, each candidate model is represented as a tree structure, where internal nodes correspond to mathematical operations (such as addition, multiplication, or sine), and leaves correspond to variables or parameters. The space of models is explored through three types of stochastic moves: node replacement, root addition/removal, and elementary subtree replacement. these moves enable transitions between any tow expressions, ensuring that the entire model space is eventually accessible.

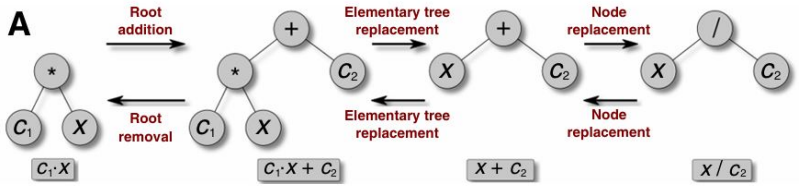


Figure 1.2: Examples of moves, extracted from [6]

At each step, a new expression f_f is proposed from the current expression f_i using one of the mentioned moves. The proposed move is accepted with probability:

$$p_{\text{accept}}(f_\mathrm{i} \to f_\mathrm{f}) = \min\left\{1, \frac{p(f_\mathrm{f} \mid D)\, g(f_\mathrm{i} \mid f_\mathrm{f})}{p(f_\mathrm{i} \mid D)\, g(f_\mathrm{f} \mid f_\mathrm{i})}\right\} \tag{1.7}$$

where $p(f|D)$ is the studied posterior plausibility of the expression and $g(f_\mathrm{i}|f_\mathrm{f})$ is the distribution of movement proposal [11]. The posterior is approximated using the previous expressions (1.2) and a prior which was obtained over expressions learned from a corpus of real-world scientific formulas, encoded via a maximum-entropy model.

To improve convergence and avoid local optima, the sampling is enhanced with parallel tempering: multiple MCMC chains run at different "temperatures", allowing for both exploration and exploitation of the expression space. Lower-temperature chains focus on accurate models, while higher-temperature chains promote structural diversity. Periodic swaps between chains facilitate a more robust and comprehensive sampling of plausible models.

This approach allows the system not only to identify the single most plausible expression (the maximum a posteriori model), but also to average over multiple plausible models when making predictions, resulting in a more accurate an reliable characterization of the underlying system.

Finally, to define a meaningful prior over mathematical expressions, the BMS learns from an empirical corpus of 4080 mathematical expressions extracted from Wikipedia pages categorized as "List of scientific equations named after people". Since Wikipedia expressions often lack context and can be ambiguous (e.g. $x^a + x^b$ might denote a vector sum or a sum of powers), a custom parsing algorithm was developed. This parser converts LaTeX-style formulas into symbolic expressions using the Python library `Sympy`, applying heuristics to resolve ambiguities [12]. Manual verification of randomly sampled subsets showed a parsing accuracy above 95%.

Once the expressions were parsed, the system constructed a prior over the expression space using a maximum entropy model. Specifically, it aimed to reproduce the empirical aver-

age and variance of the number of operations (e.g., sums, product, trigonometric functions) per expression. These statistics were captured via an exponential random graph model-like formulation, where the prior probability of an expression f_i is defined as

$$\log\left(p(f_i)\right) = \sum_{o\in\mathcal{O}} \left(\alpha_o n_o(f_i) + \beta_o n_o^2(f_i)\right), \tag{1.8}$$

where $\mathcal{O}$ is the set of all operations ($\mathcal{O} = \{+, \cdot, \exp, \ldots\}$), $n_o(f_i)$ counts how many times operation o appears in expression f_i, and $\alpha_o,\ \beta_o$ are hyperparameters fitted iteratively using MCMC, [7, 2], so that the generated expression corpus matches the empirical statistics.
This corpus-drive prior captures human-like expectations about formula structure and complexity, guiding the search away from overly complex or unnatural expressions.

1.3 Objectives

One of the main difficulties of the previous approach is that, despite the benefits that the MCMC method may bring us for sampling the expression space, it is not efficient [8]. In this project, we propose a novel method to sample the space of closed-form mathematical expressions in a more efficient way. In order to measure the "successfulness" of this approach, among other measurements, the probabilities of generating expressions between the BMS and the new approach will be compared, aiming for the best similarity between them.

One of the greatest challenges we face is the implementation of this new system. Efficiently sampling the space of closed-form expressions is a highly non-trivial task, not only due to the combinatorial explosion of possibilities, but also because of the complex structure of the underlying space. From the perspective of *information geometry*, this problem can be interpreted as finding new ways to traverse the manifold of probability distributions defined over the space of mathematical expressions, [13]. In this context, each point on the manifold corresponds to a particular distribution over expressions, and the geometry of this space plays a crucial role in determining how we can move efficiently from one distribution to another.

The main goal of this work is to develop computational tools for generating mathematical expressions that accurately fit a given dataset, while ensuring that the generation process follows a principled probability distribution. In order to address this challenge, and building on the theoretical framework discussed above, my tutors and I have defined a set of specific objectives that guide the development and evaluation of the proposed approach:

- Understand and deepen my knowledge in the problem of symbolic regression.

- Learn the foundations of the Bayesian Machine Scientist and how to use it.
- Propose, implement and study a new structure (the probability tree) to navigate in a new way through the model space.
- Evaluate the performance of the previous structure.

Chapter 2

Results

This chapter is devoted to the explanation of both the theoretical and practical results that I obtained during the project.

2.1 Theoretical Results

2.1.1 Probability Trees

The new approach is based on what we call a *probability tree*. It consists of a binary tree in which each node, instead of containing a single operation/variable/parameter, contains a map (when coding this will be referred as a dictionary) that will assign a probability to each operator (the term includes operations, functions, variables and parameters).

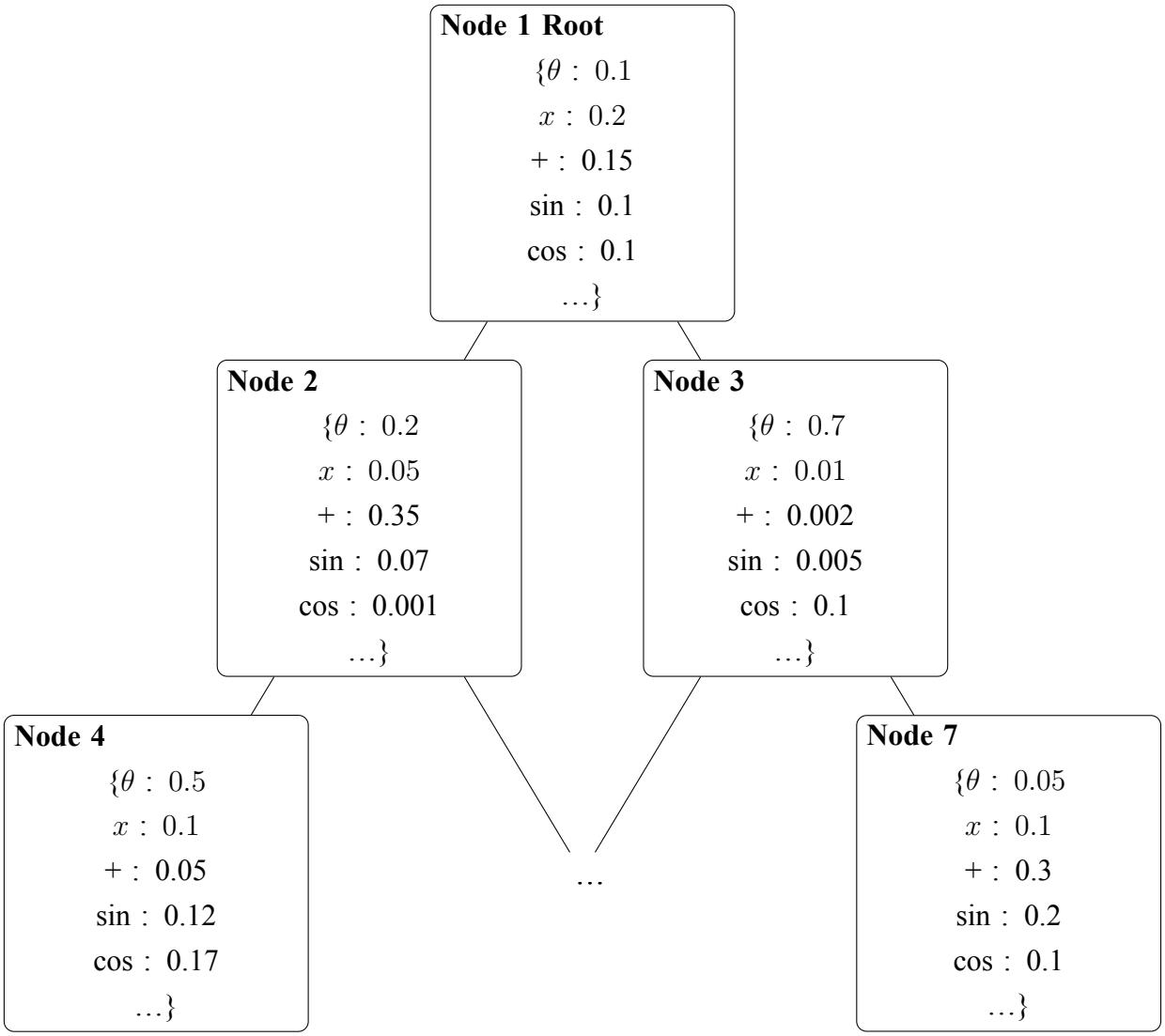


Figure 2.1: Probability tree structure example

This structure can offer some advantages with respect to other approaches. First, a more rapid exploration of the sample space. Having trained (see the following subsection) the probability tree we can traverse it by preorder to generate expressions: in each node we pick a random operation with the probability assigned for that operation in the node. This way we are making sure that the operations that are generated resemble in some way the ones that have appeared in the training corpus.

Secondly, with this implementation, a sense of "hierarchy" is inferred in the structure, because by adjusting the weights (probabilities) of operations in the nodes we can "penalize" the appearance of certain operations before others. For example, when trying to extract a physical law, it is more likely to have an expression such as $\sin(x) + \cos(x)$ rather than

$\sin(\cos(x+x))$.
Finally, function sampling is fast, since it can be easily seen by the recursivity of the traverse method that is, at most, $O(\log(\#\text{nodes}))$.

2.1.1.1 Training

As it was previously stated, this approach aims to explore the sample space in a more efficient way while preserving the generality with which the MCMC implemented in the BMS navigates it. The idea that we followed was to train the tree with expressions/models sampled from the BMS while performing a symbolic regression on a given some dataset (developed in 2). Without diving deeper in the specific aspects of the implementation, we will now focus on the core concepts behind the training of the probability tree.
First, the quality function that we chose to improve during the training process was the Kullback-Leibler divergence.

Definition 2.1 (Kullback-Leibler Divergence)**.** The *Kullback-Leibler* divergence (KL divergence) is a measure of how much different is a probability distribution with respect to another. Formally, it compares two probability distributions: a "true" one P and an approximated "approximated" distribution Q. For discrete distributions it can be computed as

$$D_{\mathrm{KL}}(P||Q) = \sum_{x\in\mathcal{X}} P(x)\log\left(\frac{P(x)}{Q(x)}\right), \tag{2.1}$$

and for continuous distributions

$$D_{\mathrm{KL}}(P||Q) = \int_{-\infty}^{\infty} P(x)\log\left(\frac{P(x)}{Q(x)}\right)\,\mathrm{d}x. \tag{2.2}$$

This quantity measures the lost information when Q is used to approximate P.

Remark 2.1. It is interesting to note the following about KL divergence:

- $D_{\mathrm{KL}}(P||Q) \neq D_{\mathrm{KL}}(Q||P)$. It is not a symmetric measure, therefore it cannot be considered as a distance either.
- $D_{\mathrm{KL}} \geq 0$ and is 0 if $P = Q$ almost everywhere.

For symbolic regression purposes, $P(x)$ would be the probability of generating some functions considering the whole sample space given the data[1]. Since it is impossible to sample all the space and assign probabilities to all expressions f, in this project, we use the

[1]Technically speaking $P(f_i|D)$, but for this part $P(x)$ is used to simplify the notation.

probabilities assigned via the description length by the BMS instead. Moreover, $Q(x)$ will be the probability of having some model generated by the probability tree.
So, to train the tree so that it samples similar models as with the MCMC method, we perform iterative changes to the probabilities inside the nodes seeking the minimization of the KL divergence. It is important to note that since we'll be working with the descriptions lengths and that we do not know the value of the normalization constant $p(D) = Z$. This however, does not interfere with the task itself. It will be seen in the following lines.

Proposition 2.1. Consider the distributions P, Q, where P is the distribution of the BMS including the normalization factor and Q is the distribution of the probability tree. Suppose that the minimum $D_{\text{KL}}(P||Q)$ is $D_{\text{KL, min}}$. Now we consider a modification of P, P', such that $P' = Z \cdot P$, where Z is a positive constant. Then, the minimum $D_{\text{KL}}(P'||Q)$ is $D'_{\text{KL, min}} = a + Z \cdot D_{\text{KL, min}}$, where a is a constant.

Proof. We will firstly see that $D_{\text{KL}}(P'||Q) = a + Z \cdot D_{\text{KL}}$ and then that the minimum is obtained when we have that $D_{\text{KL}} = D_{\text{KL, min}}$.
For the first part:

$$\begin{aligned} D_{\text{KL}}(P'||Q) &= \sum_{x\in\mathcal{X}} P'(x)\log\left(\frac{P'(x)}{Q(x)}\right) \\ &= \sum_{x\in\mathcal{X}} Z\cdot P(x)\log\left(\frac{Z\cdot P(x)}{Q(x)}\right) \\ &= Z\left(\sum_{x\in\mathcal{X}} P(x)\log\left(Z\right) + \sum_{x\in\mathcal{X}} P(x)\log\left(\frac{P(x)}{Q(x)}\right)\right) \\ &= Z\left(\sum_{x\in\mathcal{X}} P(x)\log\left(Z\right) + D_{\text{KL}}(P||Q)\right) \\ &= Z\cdot\log\left(Z\right)\sum_{x\in\mathcal{X}} P(x) + Z\cdot D_{\text{KL}}(P||Q) \end{aligned}$$

The first term is a constant, $a = Z \cdot \log(Z) \sum_{x\in\mathcal{X}} P(x)$, and the second term is the one we intended to get. The first part is concluded.
We shall prove the second part by contradiction. Since we know that $D_{\text{KL, min}}$ is the minimum value for P and Q then for any change in Q we would have $D_{\text{KL, min}} < \tilde{D}_{\text{KL, min}}$. Let us suppose now that $D'_{\text{KL, min}} = a + Z \cdot D_{\text{KL, min}}$ is not the minimum possible value of D'_{KL}, but

$\tilde{D}'_{\text{KL, min}} = a + Z \cdot \tilde{D}_{\text{KL, min}}$ is. Then we would have

$$\tilde{D}'_{\text{KL, min}} = a + Z \cdot \tilde{D}_{\text{KL, min}} < D'_{\text{KL, min}} = a + Z \cdot D_{\text{KL, min}}$$
$$a + Z \cdot \tilde{D}_{\text{KL, min}} < a + Z \cdot D_{\text{KL, min}}$$
$$\tilde{D}_{\text{KL, min}} < D_{\text{KL, min}}$$

which is obviously not true. Then the minimum $D_{\text{KL}}(P'||Q)$ is $D'_{\text{KL, min}} = a + Z \cdot D_{\text{KL, min}}$. □

What we can see with this proposition is that the value $p(D)$ is not required for performing the training since a minimum by modifying the probability tree will be obtained. Rephrasing: the task of minimizing $D_{\text{KL}}(P, Q)$ is equivalent to the task of minimizing $D_{\text{KL}}(P', Q)$.

2.1.1.2 Algorithm for Training

The implementation of an algorithm for the training for the probability tree is as introduced before: minimizing the KL Divergence by applying changes to the probabilities of different nodes.

Algorithm 1 Training of the probability tree: pseudocode KL divergence minimization

Input: The probability tree T, the Bayesian Machine Scientist B, and a dataset of models D, number of iterations n.

```
D_KL ← Get D_KL(B, T, D)
for i = 0; i ≤ n; i = i + 1 do
    T' ← T.copy
    T' ← T'.modify_probability_rand_node
    D'_KL ← Get D_KL(B, T', D)
    if D'_KL < D_KL then
        T ← T'
        D_KL ← D'_KL
    end if
end for
return T, D'_KL
```

For each iteration, a copy of the probability tree is created, one of its nodes has its probabilities modified and if the KL divergence with this modified tree is inferior to the current one, both the original tree and the divergence are updated. The implementations of how the probability is modified will be explained in the methodology part (2). As for the function

that computes the divergence, its implementation varies depending on the dataset format. The next algorithm is a general implementation.

Algorithm 2 Get D_{KL}

Input: The BMS B, The probability tree T and a dataset of models generated by the BMS D. The file D has the following format: for every line there is just one model f_i.

$D_{\mathrm{KL}} \leftarrow 0$
for $f_i \in D$ **do**
  $\mathcal{L} \leftarrow B$.get_description_length(f_i)
  $P \leftarrow e^{-\mathcal{L}}$
  $Q \leftarrow T$.get_probability(f_i)
  $D_{\mathrm{KL}} \leftarrow D_{\mathrm{KL}} + P\log\left(\frac{P}{Q}\right)$
end for
return D_{KL}

Remark 2.2. Note that for obtaining the true value of the KL Divergence, we would need to sum over all the possible models. This is impossible in practice. So, the numerical results that we obtain are approximations of the true value.

2.2 Experimental Results

2.2.1 Use of the Bayesian Machine Scientist

In this part the objective was the familiarization with the functioning of the BMS and its libraries. It is important to note that the algorithm is available online at https://bitbucket.org/rguimera/machine-scientist/src/no_degeneracy/. This repository was downloaded and it is ready to use via Jupyter Notebooks. For a symbolic regression task, the procedure needed to carry it out is the following:

1. Import necessary libraries.
2. Import/Generate the data.
3. Initialize the BMS
4. Sample expressions keeping track of the model with lowest description length.
5. Print the results.

Each step will be explained in the following paragraphs.
First, we import the necessary libraries. The libraries contain all the necessary tools and information to carry out all the process in a simple and clean way, such as functions that perform a specific task or that transform a file into a treatable format. The most important libraries and their purpose in the BMS are introduced in the following table.

Library	Purpose in the BMS
Numpy	Allows to perform fast and efficient operations, specially with arrays. It includes manipulation of multidimensional arrays, vectorized operations, mathematical functions (e.g. `np.sin`) or data generation. For example, it is used for performing fast summations of squared errors, the use of mathematical functions or assigning infinite values to some variables (step necessary in some algorithms that are internally used).
Pandas	Manipulation and analysis of structured data. Write and read files, DataFrame manipulation, filtering and selection. It has several more utilities, yet they are not used in the algorithm. For example, the input dataset is loaded as a Panda's DataFrame, or the set of constants, which is created as vector.
Sympy	For symbolic calculations, such as performing the simplification of a function, or telling if two functions are actually the same or not (critical task for the sampling of models). For example, the trees that the BMS uses are converted into mathematical expressions via `sympify`. It also helps to convert some expression into a LaTeXformat, which for some cases, such as plots or exporting information, is very useful.
Matplotlib	Visualization of results: generating plots (in this case the evolution of the description length and the comparison between the predictions of the best model and the actual values).
MCMC	Refers to the Markov Chain Monte Carlo algorithm used for probabilistic inference and sampling in the model. That is, the method used for sampling functions in a representative way to reasonably navigate through the model space. It also contains other core structures that are used through the project: the implementation of the data structures of the tree and its nodes, the functions associated to the possible movements in the tree such as elementary tree replacements or root replacement. It also contains the method (among others) that fits the used parameters.
Parallel	It contains the class for parallel tempering, which allows to introduce a controlled randomness in the algorithm, avoiding getting stuck in local minima of description length. Moreover, it contains the instruction for performing "steps" (elementary actions) with the trees.
Read_prior_par	Reads the file where the prior values and parameters (and hyperparameters) are stored. This is used to then compute the probability of a model by 1.8. This one forms parts of a bigger library, called Fit_prior.

Table 2.1: Most important libraries

We have not included some other libraries since they take care of minor aspects, such as handling certain kind of errors, displaying a progress bar, etc.
The code for importing the necessary libraries can be seen in the following paragraph.

```
import sys
import numpy as np
import pandas as pd
import warnings
warnings.filterwarnings('ignore')

import matplotlib.pyplot as plt
from copy import deepcopy
from ipywidgets import IntProgress
from IPython.display import display

sys.path.append('./')
sys.path.append('./Prior/')
from mcmc import *
from parallel import *
from fit_prior import read_prior_par
```

Now, data can be obtained in two ways: reading it from an external file, such as an spreadsheet or a CSV (Comma Single Value) file, or generating it according to some predefined procedure. The important thing is that the data has to be converted into two Panda's Dataframes: one containing the independent variables `x` and one containing the dependant variable `y`. For example, in the following code we import data from a spreadsheet named `gravitation_data_example.xslx` and we set `r`, `m1`, `m2` as independent variables and `F` as the dependent variable.

```
XLABS = [
    'r',
    'm1',
    'm2',
]
raw_data = pd.read_excel('./gravitation_data_example.xlsx')
x, y = raw_data[XLABS], raw_data['F']
x.head(10)
```

```
XLABS = [
    'r',
    'm1',
    'm2',
]
raw_data = pd.read_excel('./gravitation_data_example.xlsx')
x, y = raw_data[XLABS], raw_data['F']
x.head(10)
```

	r	m1	m2
0	963	19151510	14716982
1	575	3417254	7155752
2	494	7350373	4704939
3	465	15539602	2453943
4	384	10863679	19684830
5	305	19968878	4172071
6	466	3239195	7092104
7	837	6412228	9930312
8	922	9199724	4980842
9	234	18711619	17760569

Figure 2.2: Example data importation

Once we have prepared the data, it is time to initialize the BMS. This involves two aspects: to read the file of priors that are going to be used (the one that helps us to compute the probabilities of models via the use of fitted parameters and hyperparameters from the Wikipedia corpus) and initializing the parallel tempering structure that the BMS uses.

```
# Read the hyperparameters for the prior
prior_par = read_prior_par('./Prior/final_prior_param_sq.named_equations.
    nv3.np3.2017-06-13 08_55_24.082204.dat')

# Set the temperatures for the parallel tempering
Ts = [1] + [1.04**k for k in range(1, 20)]

# Initialize the parallel machine scientist
pms = Parallel(
    Ts,
    variables=XLABS,
```

```
    parameters=['a%d' % i for i in range(13)],
    x=x, y=y,
    prior_par=prior_par,
)
```

Having made all the preparations, the BMS proceeds to sample models by doing an iterative use of the functions `mcmcm_step` and `tree_swap`. For each iteration, we compute the description length and store the model with the lowest one (the one that best describes the data). The number of iterations is customizable via the variable `nstep`.

```
# Number of MCMC steps
nstep = 5000

# Draw a progress bar to keep track of the MCMC progress
f = IntProgress(min=0, max=nstep, description='Running:') # instantiate the bar
display(f)

# MCMC
description_lengths, mdl, mdl_model = [], np.inf, None
for i in range(nstep):
    # MCMC update
    pms.mcmc_step() # MCMC step within each T
    pms.tree_swap() # Attempt to swap two randomly selected consecutive temps
    # Add the description length to the trace
    description_lengths.append(pms.t1.E)
    # Check if this is the MDL expression so far
    if pms.t1.E < mdl:
        mdl, mdl_model = pms.t1.E, deepcopy(pms.t1)
    # Update the progress bar
    f.value += 1
```

Finally, we print the best model and plot the evolution of the description length.

```
print('Best model:\t', mdl_model)
print('Desc. length:\t', mdl)
plt.figure(figsize=(15, 5))
plt.plot(description_lengths)
plt.xlabel('MCMC step', fontsize=14)
plt.ylabel('Description length', fontsize=14)
plt.title('MDL model: $%s$' % mdl_model.latex())
plt.show()
```

```
Best model:      ((-(m1) / ((r / ((log(_a5_) ** 2) + _a5_)) * (_a4_ / -(m2)))) / r)
Desc. length:    -1054.78069579807
```

Figure 2.3: Best model obtained

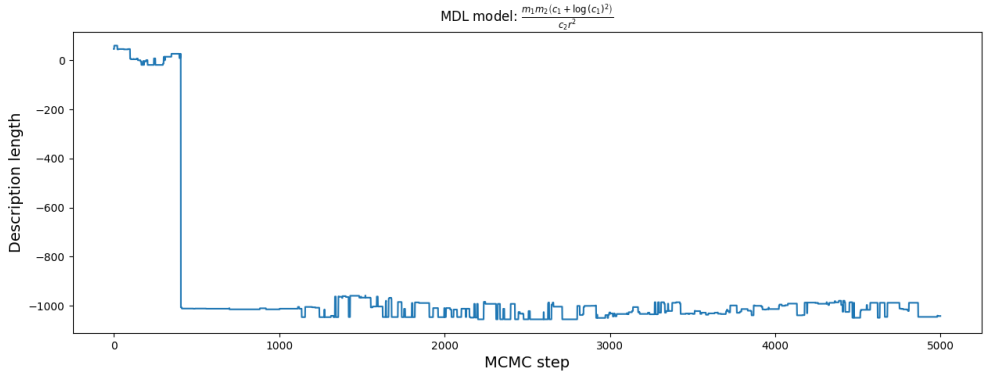


Figure 2.4: Description length progression

We can study the results now. In the case of the example, the BMS has correctly discovered the gravitation law ($F = G \cdot m_1 \cdot m_2 / r^2$). The correct structure was most likely detected at approximately the 500th step, because of the sudden step decrease in the description length. Then some minor variations of the model are proposed up to the end of the sampling process. These variations arise due to the parallel tempering, the random tree changes that may have been proposed, and also to adjust or create another combination of parameters.

It is important to know that there are more functionalities that the BMS has, such as the possibility of providing a model to start from, or the possibility of running multiple BMSs on several datasets at the same time.

This first experimental part consisted on getting used to the environment and, more importantly, understanding what was happening in the code, so that meaningful ideas and conclusions can later be extracted. So I experimented with the algorithm on several datasets, understanding its strengths and limitations and knowing its most important functions.

2.2.2 Datasets and Data Generation

This part of the project is devoted to the creation of an experimental corpus that was later used to train the new probability tree structure. Using the BMS as a reference, we created a corpus of models and computed their description lengths. The purpose of this part is to be able to "teach" the new structure which expressions should be "penalized" or "awarded" depending on the dataset that was used.

The dataset that we used for this project is a popular benchmark in the symbolic regression literature, the Nguyen dataset [10]. More concretely, we used a specific part of this dataset. It is composed of the following datasets:

Name	Expression	Dataset
Nguyen-1^c	$3.39x^3 + 2.12x^2 + 1.78x$	$\mathcal{U}(-1, 1, 20)$
Nguyen-5^c	$\sin(x^2)\cos(x) - 0.75$	$\mathcal{U}(-1, 1, 20)$
Nguyen-7^c	$\log(x + 1.4) + \log(x^2 + 1.3)$	$\mathcal{U}(0, 2, 20)$
Nguyen-8^c	$\sqrt{1.23x}$	$\mathcal{U}(0, 4, 20)$
Nguyen-10^c	$\sin(1.5x)\cos(0.5y)$	$\mathcal{U}(0, 1, 20)$

Table 2.2: Subsection of the Nguyen dataset that was used. $\mathcal{U}$ means that the data is generated uniformly in the specified range

To each point of all the datasets, we applied a random Gaussian error with standard deviation $\sigma = 0.1$. With this deviation, which is not negligible since the values that all these functions take in their respective ranges are small, the BMS is forced to sample a wider variety of expressions and also has to compute the description length. Thanks to this part, a broader list of expressions can be sampled and provides a more comprehensive input to train the new structure. It is important to note that the σ value was not chosen at random, but as a result of studying at which standard deviation value the BMS was not able to produce a reasonable model [3]. We used a lower value than this bound.

As for the implementation, first, in order to introduce numerical data into the BMS algorithm, we modified some parts of the code from the previous example. The data creation/import would stay as:

```
# one variable version
XLABS = ['x']
xp = np.random.uniform(0,4,20)
raw_data = pd.DataFrame({'x': xp})

def func(x):
# function declaration
    return  np.sqrt(1.23*x)
yp = func(xp)

# add gaussian error
stddev = 0.1
mean = 0
noise = np.random.normal(mean, stddev, yp.shape)
```

```
15
16 yp_mod = yp + noise
17
18 x = raw_data[XLABS]
19 y = pd.Series(yp_mod)
20
21 # two variable version
22 XLABS = ['x','z']
23 xp = np.random.uniform(0, 1, 20)
24 zp = np.random.uniform(0,1,20)
25 X,Z = np.meshgrid(xp,zp)
26
27 def func(x,z):
28     return np.sin(1.5*x)*np.cos(0.5*z)
29
30 Y = func(X,Z)
31
32 # add gaussian error
33 stddev = 0.1
34 mean = 0
35 noise = np.random.normal(mean, stddev, Y.shape)
36 Y_noisy = Y + noise
37
38 raw_data = pd.DataFrame({'x': X.ravel(), 'y': Y_noisy.ravel(), 'z': Z.
       ravel()})
39 x = raw_data[XLABS]
40 y = raw_data['y']
```

Additionally, we created an auxiliary library to name and store data files in a systematic and organized way, named `file_creator`. Using this library, we create the file where the data will be stored before starting to sample expressions.

```
1 resultsDir = "Results_Leo/New_Datasets/" # provisional folder where
      stored
2 filepath = file_namer(func, parameter_path, resultsDir, mean, stddev)
```

Finally, we introduced another modification to write the progress in the file.

```
1 # Number of MCMC steps
2 nstep = 20000
3
4 # Draw a progress bar to keep track of the MCMC progress
5 f = IntProgress(min=0, max=nstep, description='Running:') # instantiate
      the bar
6 display(f)
```

```
7
8  # MCMC
9  description_lengths, mdl, mdl_model = [], np.inf, None
10 #Modification to write in the file
11 with open(filepath, 'w') as file:
12     for i in range(nstep):
13         # MCMC update
14         pms.mcmc_step() # MCMC step within each T
15         pms.tree_swap() # Attempt to swap two randomly selected
   consecutive temps
16         # Add the description length to the trace
17         description_lengths.append(pms.t1.E)
18
19         #Save information in the file
20         file.write(f"{i} || {round(pms.t1.E, 3)} ||{pms.t1.pr(show_pow=
   True)}||")
21         #Before printing the constants, let us improve the format in
   which they appear
22         inner_dict = pms.t1.par_values['d0'] #extraiem el diccionari dins
    de d0
23         # We'll create a new dictionary with a more convenient format
24         formatted_dict = {key.strip('_'): round(value,3) for key, value
   in inner_dict.items()}
25         dict_string = ', '.join([f"{key} : {value}" for key, value in
   formatted_dict.items()])
26         file.write(f"{{{dict_string}}}\n")
27
28         # Check if this is the MDL expression so far
29         if pms.t1.E < mdl:
30             mdl, mdl_model = pms.t1.E, deepcopy(pms.t1)
31         # Update the progress bar
32         f.value += 1
```

With all of the above, we generated files containing lots of the models. The format for each line in these files is shown in the following figure.

```
1345 || 35.222 ||((pow2(((x + (_a6_ ** x)) ** _a1_)) * ((_a12_ * (_a6_ * x)) + _a2_)) + _a4_)||{a1 : 1.64, a2
: 4.076, a0 : 1.0, a3 : 1.0, a5 : 1.0, a7 : 1.0, a8 : 1.0, a9 : 1.0, a10 : 1.0, a11 : 1.0}
1346 || 39.171 ||(((_a3_ * _a2_) / (_a4_ ** x)) + ((_a7_ + _a8_) + (_a11_ * (_a2_ ** (x * _a3_))))))||{a2 : 0.2
```

Figure 2.5: Example of sampling file of models

Taking into account the delimiters we have, the step number, the description length, the model (function) and the values of its parameters.

Via the previous code, we created five files that were used later. In the repository, they can be found inside the folder named `Results`. Per each file, about 20×10^3 models were sampled. Nevertheless, lots of these models are the same expression. We removed these repetitions, which left us with files about one ninth the size of the original.

We finished this part once these files were generated. However, I had to revisit the code for this part several times in the later stages of the project in order to double-check some results or to evaluate modifications of the tree structure.

2.2.3 Probability Tree Implementation

In this part, we introduce the idea behind the the probability tree implementation, including its main attributes and functions, and auxiliary data structures that were needed to build it.

As previously mentioned, the aim was to build a structure such as the one in 2.1. Before coding it, I first listed the essential components the structure required. In my case, these were:

- Probability Nodes: the structure from which the tree is built. These nodes contain (each one) the dictionary of probabilities.
- A method to initialize a tree. This is essential to later initialize instances of this class.
- A method to compute the probability of a given expression using the tree.
- A method to sample expressions with the adjusted probabilities via the probability tree.

Thanks to these main components, we could add other functions and attributes during the process, creating a more complete structure. In the following two subsections we detail the content of the probability nodes and the probability trees.

2.2.3.1 Probability Nodes

Probability nodes (`ProbNode` in the code) are the fundamental component of the tree. This class must contain the dictionary of probabilities as an attribute, methods to modify the probability of the dictionary, a set of variables and parameters that are used, etc. I will divide the explanation of the content of the probability node into two parts: attributes and methods. For the part of attributes, we have:

- Parent and offspring: Since the node is part of the tree, it must contain references to both the preceding node and the next nodes so the tree can be traversed. The maximum allowed length of the offspring is 2 since we have a binary tree structure.

- Operation list, variable list and parameter list: These lists contain the set of possible operations ($\mathcal{F} = \{+, -, \ldots\}$), variables ($x, y, z, \ldots$) and the parameters ($\theta = \{\theta_1, \theta_2, \ldots\}$). All of these lists can be initialized with default values or custom values can be assigned.

- Operation and variable probability list and parameter probability list: When a node is created I wanted to have variables and operations with the same probability assigned, and the parameters with the same probabilities among them. That is why I created two dictionaries of probabilities, one that contains all the operations and variables (each element with the same probability at the beginning) and one that contains all the parameters (each element with the same probability at the beginning).

- Variable of the probability of operations: With this parameter the user can initially assign more probability to the parameters or the variables and operations.

- Probability list: Combining the previous probability lists and the parameter for the probability of operations, we crated a global probability dictionary.

As for the functions that this class, we have:

- Initialize: Function for initializing the structure, with all the attributes above. If not specified, the value of the parameter for the probability of operations will be set to 0.5, the variables to just one and only one constant.

- Add son: Add an element (another probability node) to the offspring of a node. Since the function is internally used, there is no double check for when a third child wants to be added.

- Generate the probability list: In case the parameter for the probability of operations is set to another number this function is called and creates the complete probability list accordingly.

- Random modification: This function is essential for the training of the tree. It randomly selects an element from the probability list and performs the following operation. If the selected element is a parameter with probability p, then its probability is reassigned to

$$p_{\text{par}} \leftarrow \max\{\varepsilon, p_{\text{par}} + \eta\}, \tag{2.3}$$

where $\eta \sim \mathcal{N}\left(\mu = 0, \sigma = \frac{0{,}014}{|\theta|}\right)$, and $\varepsilon = 10^{-6}$. This value transformation makes a small change in the probability of a variable (further referred as *Gaussian step*, for

simplicity). In case the value $p + \eta$ is negative, since there cannot be negative probabilities, a very small number is assigned. If the randomly chosen element is a variable or an operation, the same operation is performed but with $\eta \sim \mathcal{N}(\mu = 0, \sigma = 0.01)$. The values of these standard deviations have been fine-tuned so that a high number of changes is accepted in the process of minimizing the KL Divergence. Once the probability has been modified, the dictionary of probabilities is normalized. Furthermore, it is also interesting to note that there was another approach implemented before this one. Its explanation and results can be found in A.

- Node copy: This function creates an exact replica[2] of a probability. It is used as an auxiliary function in the probability tree, since a function for copying trees is defined there.

- Node run: Given a probability node this function traverses from this node to the bottom of the tree. It makes us of recursion and serves as an auxiliary function for the method for traversing a probability tree.

- Pick random operation: This function picks a random element from the probability list taking into account the probabilities of the dictionary.

- Generate a random expression: Given a node, it traverses the probability tree from the node itself to the bottom, picking operations using the previous function and returns an expression in a BMS-friendly format. This function uses recursivity.

2.2.3.2 Probability Tree Content

This class (`ProbTree` in the code) uses the probability nodes and can optionally contain a BMS as an attribute[3]. As for the attributes this class has:

- Tree: The BMS that was previously mentioned. By default it is not initialized.

- A list of variables and parameters: In case the probability tree needs to create new nodes, these lists will be used as a reference.

- A probability node: this will serve as the root of the tree. If specified, a probability tree can be grown from a given probability node.

[2] Take into account that in the case of Python, this is a shallow copy: it duplicates the node's content but not the objects referenced by its pointers.

[3] It is not compulsory to initiate it, it was added as an attribute to ease some calculations, such as when comparing probabilities.

- A parameter for the maximum size of the tree: It determines how many nodes the tree will initially have. By default this is 100 nodes. As we will see in the functions part, the class automatically adds probability nodes if needed (e.g. when computing the probability of an expression that has nodes that go deeper than the ones the tree has).

And for the functions:

- Initialize: Function for initializing the structure. It sets the parameters, variables and a root probability node (if no node was provided). This tree is created with a default size of 100 nodes. In order to do so it makes use of the next function.

- Tree builder: Auxiliary function to build the tree. Since we have a binary tree and a maximum number of nodes. It creates the closest number of nodes to the specified maximum M so that the tree is complete and symmetric in the beginning. To do so, knowledge in data structures is used: first of all, the largest number of nodes n is computed, that is $n = \max\{k \in \mathbb{Z} \, | 2^k - 1 < M\} = \lfloor \log_2(M+1) \rfloor$. Once this number is computed, a list of nodes is created, and it will be used to establish the parent-child relationship, taking advantage that if a node is in position k, its offspring is located in position $2k$ and $2k+1$. Finally, the first node of the list -which will be root- is returned.

- Tree run: If the instance contains an expression (i.e., a binary expression tree), either provided directly or embedded in the BMS structure, this method traverses the associated probability tree and collects the probability values assigned to each operation in the expression. The traversal follows the structure of the expression, which is represented as a binary tree. At each node, it extracts the probability corresponding to the operation found at that position. If a corresponding node in the probability tree does not yet have children, the method dynamically expands it by adding default child nodes. The result is a list of probabilities, one for each operation in the expression, ordered according to the tree traversal.

- Probability of the tree: Calls the previous function and multiplies all the elements in the list, returning the probability of the expression that is stored in the BMS attribute.

- Probability given an expression: It computes the probability of a given expression, not necessarily the one in the BMS attribute.

- Modify a random probability: Picks a random node (considers the probability tree in a list-form and then picks a random node, so that the choices do not depend on the level

of the tree) and uses the function "Random modification" on the selected node. This function is fundamental in the training process of the tree.

- Random function: This function generates a random function using the tree and taking into account the probability list of each node.
- Tree copy: Creates an exact replica (not a pointer, but a new instance of a probability tree with the same characteristics) of the current tree. This function is useful in case a random modification of a tree is rejected in the training process, so that there is a way to "revert" that change.

2.2.4 Probability Tree Training and Testing

Before training the trees, we add another element to the training datasets. As it was previously said, the training datasets contain expressions sampled by the BMS. In general, the evolution of the expressions that are sampled by this algorithm usually goes "on the right path", that is, variations of the previous models that we already know have a low (potentially good models) description lengths. However, in order to create a more representative dataset to train the tree, expressions that are "very bad" candidates should be added too [5]. This way the tree also learns to penalize certain operations. With this in mind, we randomly generated a set of $\sim 20 \times 10^3$ expressions with a default probability tree and, for each of the different Nguyen's datasets that we used, and calculated their description length with the BMS. Finally, the obtained results were combined with the datasets created in 2.2.2.

Now, having prepared both the structures and the datasets, the training of the trees can begin. As explained in Algorithm 1 the idea is to iteratively apply random changes on the probability tree via the "Modify a random probability" and, if the change reduces the KL Divergence, accept the change. Once this process has finished, we store the final probability tree, and then we compare the description lengths and visualize the negative logarithms[4] of the probabilities by means of a scatter plot. For this experimental design we used $N = 25 \times 10^3$ iterations to train each tree.

Remark 2.3 (About the implementation of the training process)**.** The training process is computationally slow, since it has to recompute the probabilities of $\sim 20 \times 10^3$ mathematical expressions per each iteration. Performing this task every iteration in a sequential way for 25×10^3 makes it to last even longer (training periods ranging from 1 day to 2). To overcome this, a parallel version of the KL Divergence computation was implemented. The idea behind

[4]Since the order of the values of probabilities varies a lot, it is easier to extract conclusions from a plot of logarithms rather than the raw values.

this approach is to divide the full task into smaller, independent blocks that can be processed simultaneously. Instead of evaluating all expressions sequentially, the dataset is split into batches, and each batch is assigned to a different worker. Each worker corresponds to a separate CPU core, allowing multiple parts of the computation to be carried out in parallel. This parallelization significantly reduces the total training time, as the workload is efficiently distributed across the available computational resources, [15].

This is, in essence, the idea behind *parallel processing*: breaking down a large task into smaller units that can be executed at the same time. To implement this, additional Python libraries were used—namely, `concurrent.futures`, which provides a high-level interface for parallel execution using processes, and `multiprocessing`, which allows detection and management of the available CPU (Central Processing Unit) cores. These tools made it possible to redesign the KL computation in a way that takes full advantage of modern multi-core architectures. With this alternative implementation, computing times were reduced more than 50%.

Remark 2.4. Currently, the way we modify probabilities in this algorithm follows the expression in 2.3. However, we used another approach before, and it consisted in the following: picking a random node, then pick a random element exclusively from the list of variables and functions, modifying its probability inside that list via generating a random number between 0 and 1, adjust the probabilities of the other operations and variables proportionately and then generating a probability list but assigning a random number between 0 and 1 to the "Variable of the probability of operations" (see A). The positive part about this approach is that it managed to decrease the KL Divergence considerably ($\sim 25\%$). Its main drawback is that the way the tree was learning could be considered a bit "naive". For example, a change in which increasing the probability of the operation factorial to 0.7 could be accepted (despite not appearing in any model) if it drastically reduced the probability of another operation that appeared less, such as the absolute value. Moreover, the other problem with this idea is that it would be really hard to "converge" to any state, since the changes in nodes are extremely abrupt. By using Gaussian steps instead, it is easier to control that the right operations have their probability increased and the wrong ones decreased.

In the training process, we stored and plotted the evolution of the KL divergence over all the steps (iterations), figs. 2.6, 2.8, 2.10, 2.12 and 2.14. We then plot the description length obtained by the BMS (which is the negative logarithm of the posterior) to the negative logarithm of the probabilities assigned by the probability tree. We do this for both the untrained and the trained trees, figs. 2.7, 2.9, 2.11, 2.13 and 2.15.

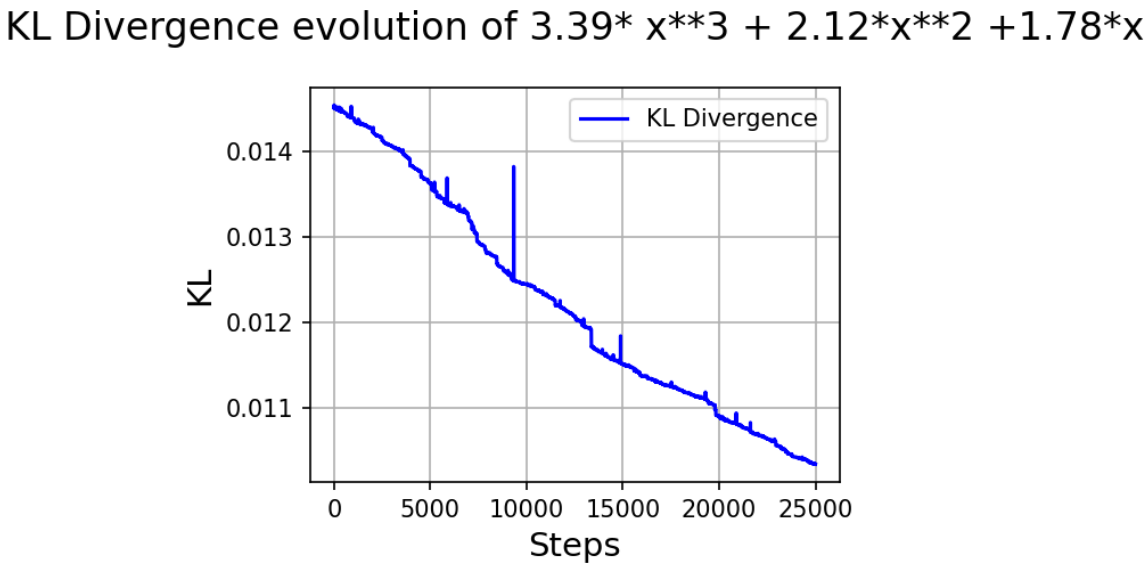


Figure 2.6: KL Divergence evolution of $3.39x^3 + 2.12x^2 + 1.78x$

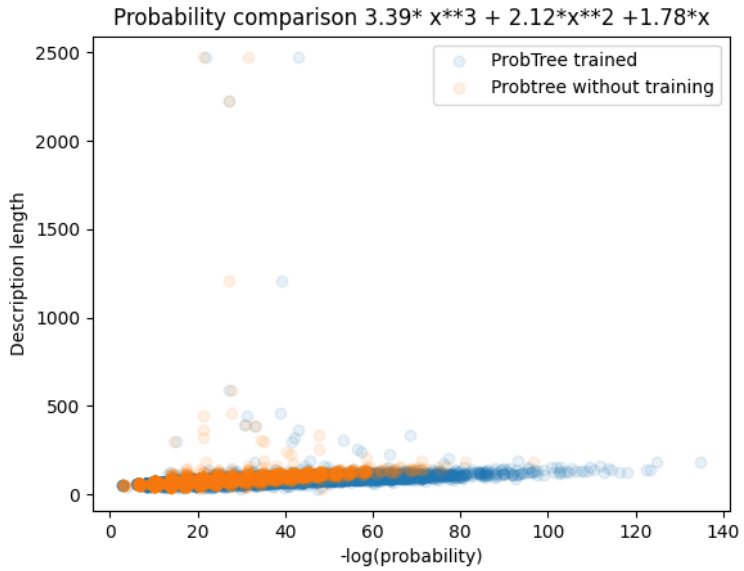


Figure 2.7: Probability comparison $3.39x^3 + 2.12x^2 + 1.78x$

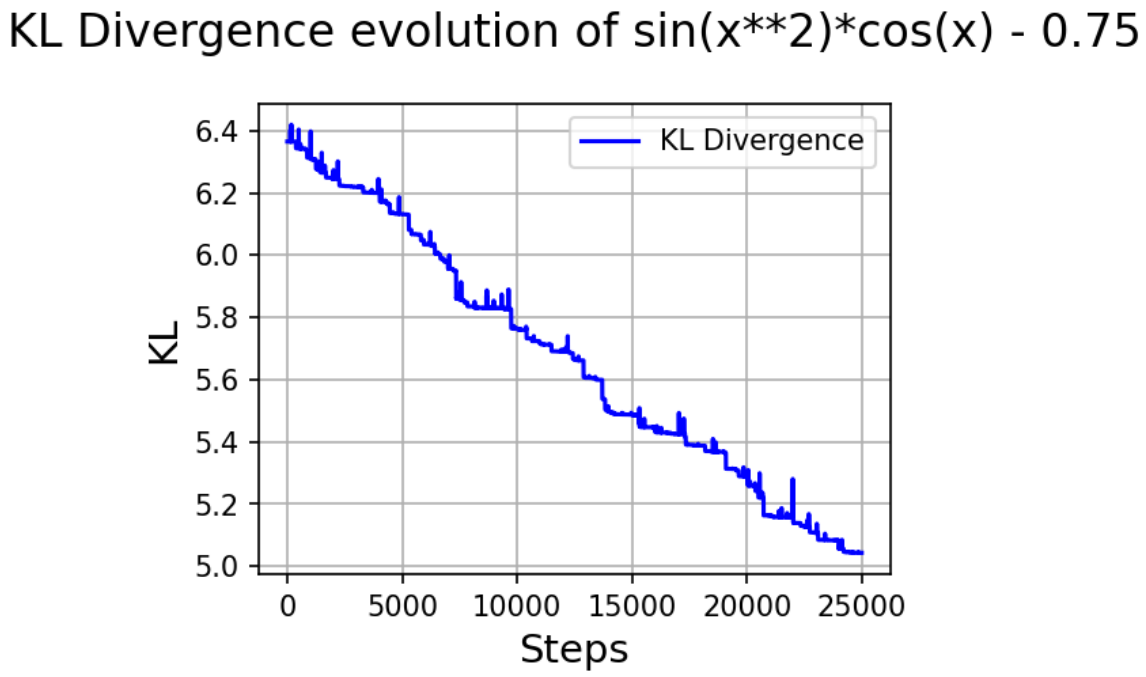


Figure 2.8: KL Divergence evolution of $\sin(x^2)\cos(x) - 0.75$

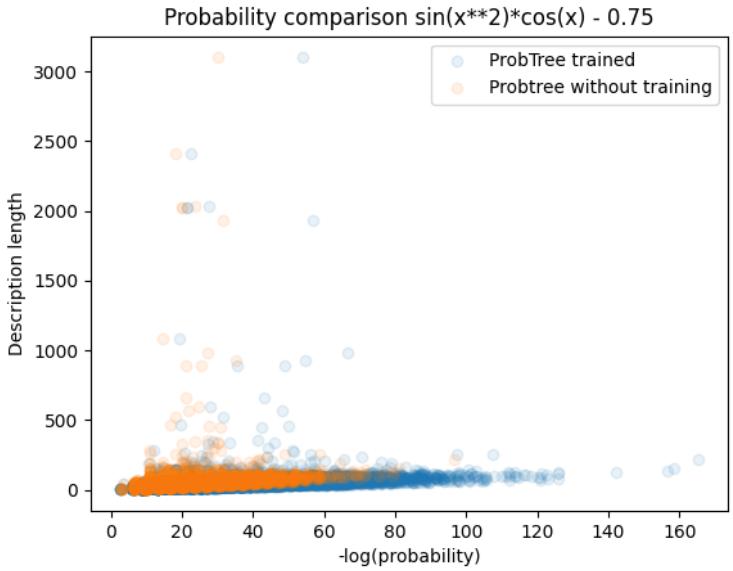


Figure 2.9: Probability comparison $\sin(x^2)\cos(x) - 0.75$

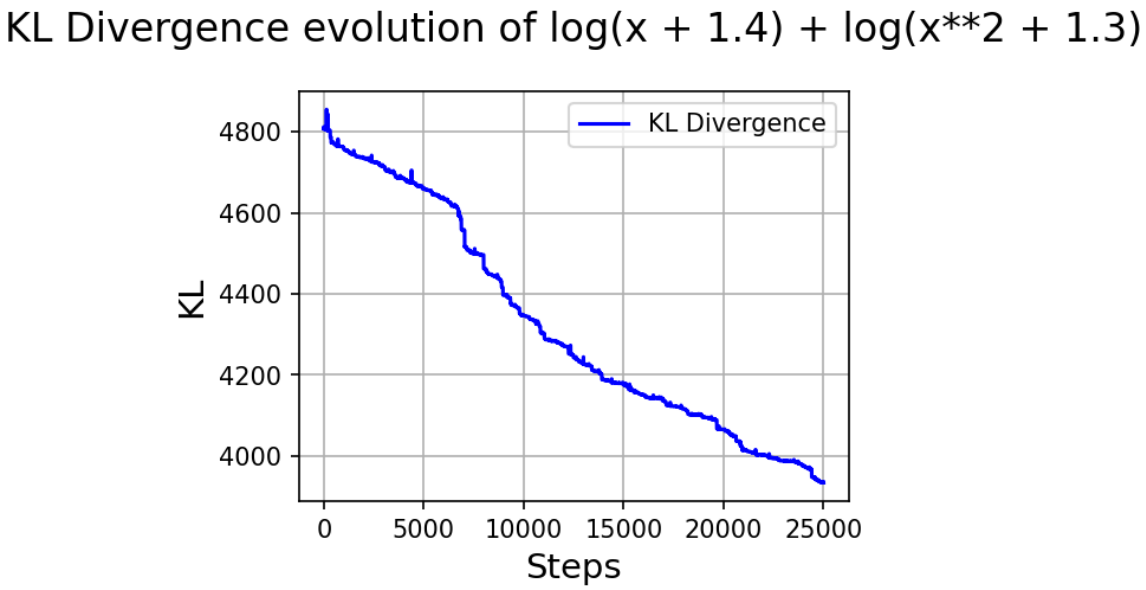


Figure 2.10: KL Divergence evolution of $\log(x + 1.4) + \log(x^2 + 1.3)$

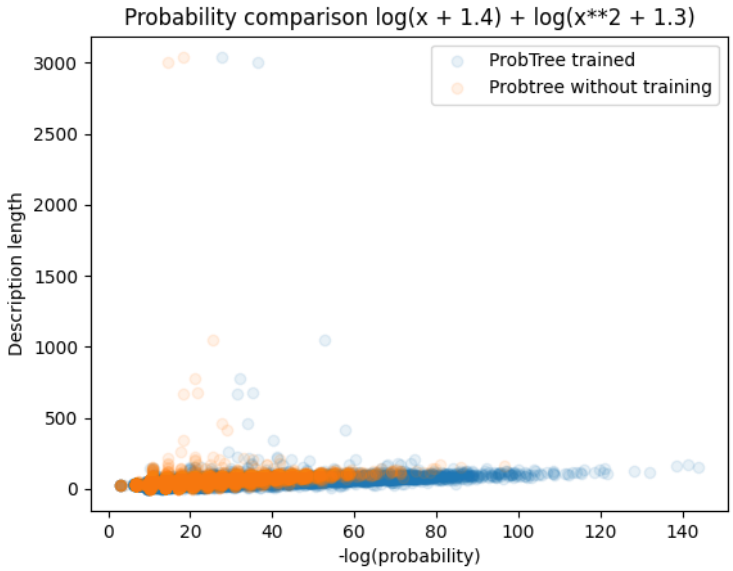


Figure 2.11: Probability comparison $\log(x + 1.4) + \log(x^2 + 1.3)$

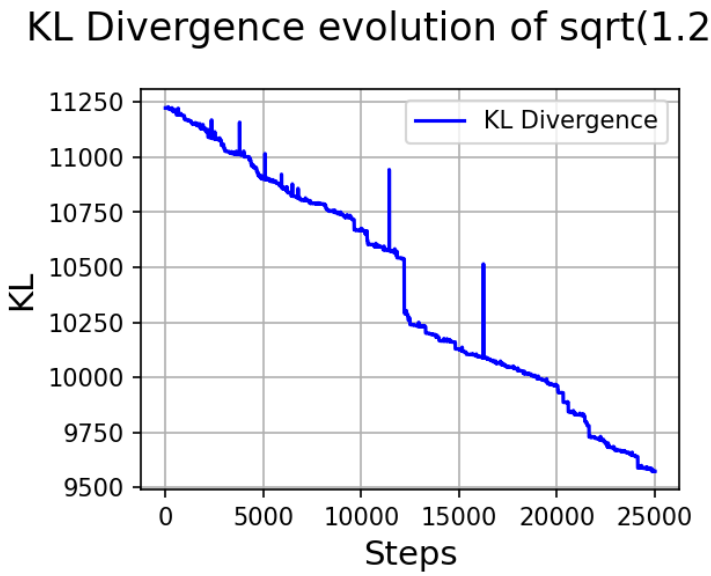


Figure 2.12: KL Divergence evolution of $\sqrt{1.23x}$

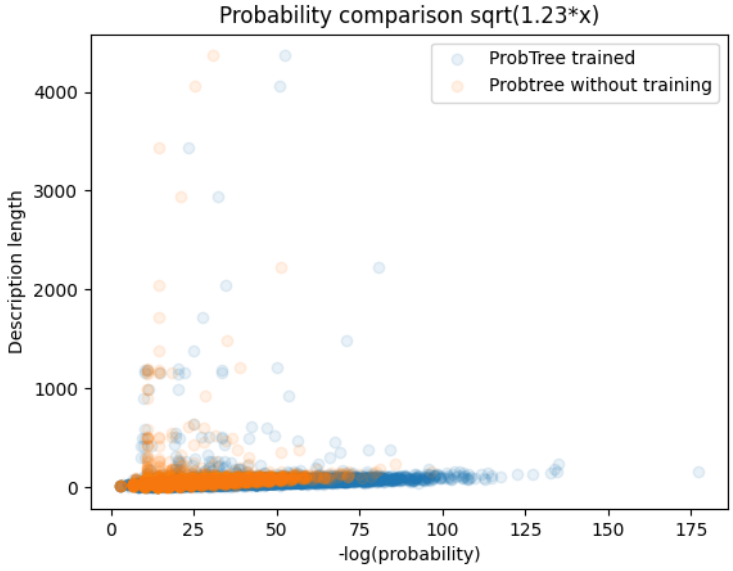


Figure 2.13: Probability comparison $\sqrt{1.23x}$

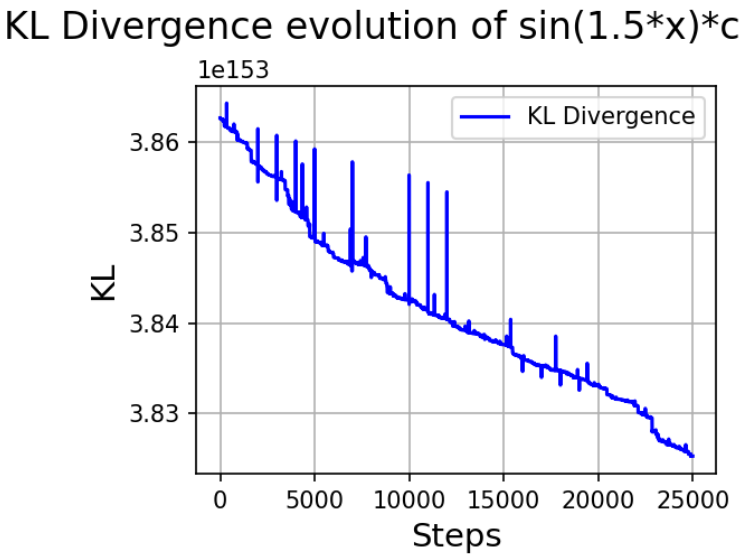


Figure 2.14: KL Divergence evolution of $\sin(1.5x)\cos(0.5z)$

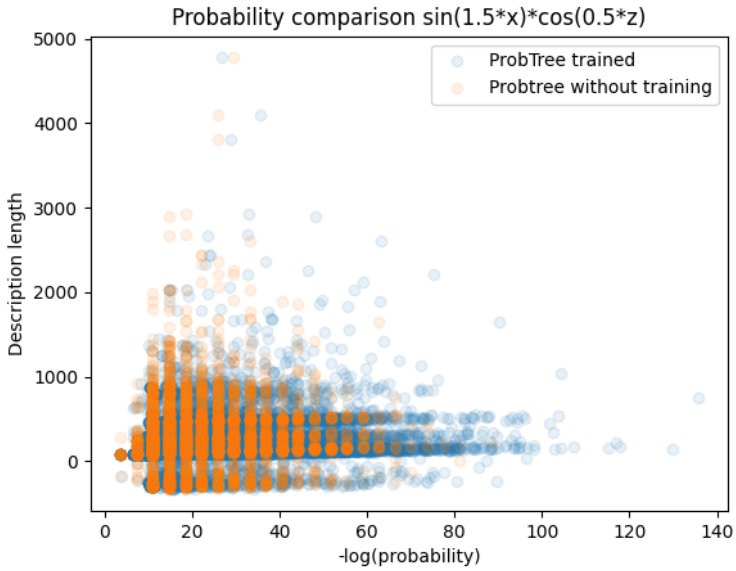


Figure 2.15: Probability comparison $\sin(1.5x)\cos(0.5z)$

2.2.5 Analysis of the Training

The analysis is divided into two parts: the evolution of KL divergence, and the comparison of the description length versus negative logarithm of the probabilities plots.

Regarding the evolution of Kullback–Leibler divergence (figs. 2.6, 2.8, 2.10, 2.12 and 2.14), we can appreciate that the implemented rule of updating probabilities described in 2.3 pro-

vides a uniform rate of improvement. In fact, by observing any of the plots, it is very likely that if we performed more iterations, we could obtain lower values of the parameter. It is also interesting to note that some "spikes" (sudden increases) in the KL divergence values have appeared during the process. This behaviour stems from the fact that there might be some probability changes applied to the upper nodes of the tree that are not a good choice, leading to these cases. This happens a lot particularly in the case of fig. 2.14. The opposite might happen as well, as it is the case of the sudden decrease in the parameter value in fig. 2.12, at step ~ 12500.

Regarding the probability comparison plots, it is first essential to understand the quantities that are being represented. In the vertical axis, we have the description lengths that the BMS has assigned to expressions generated by a probability tree. It is important to take into account that this value corresponds to the logarithm of the probability of having some expression given the data ($\mathcal{L}(f_i) = -\log\left(p(f_i|D)\right) - \ln Z$, it can be derived by manipulating the terms in equation 1.2). So, if we compute a probability using the probability tree, in order to produce a meaningful comparison, the negative logarithm of that probability must be applied, and that would constitute a proper horizontal axis to use in a visual representation. This is what we plotted in the *Probability comparison* set of figures (figs. 2.7, 2.9, 2.11, 2.13 and 2.15).

Having said that, the ideal result would be that both the BMS and the probability tree mapped the expression to the same probability. In that case, the description length and this negative logarithm of the tree would only differ in a constant term, $-\ln Z$. In other words, there would be a linear relationship between these two quantities[5]. Using simple terms, we would like to see that big/low values of description lengths are mapped, with the same rate/order, to big/low negative logarithms of probabilities values. Knowing that, then how do we know if the probability tree has learnt or not with the training process? We did this by comparing the logarithm of the probability of the untrained tree to that of the trained tree. For example, if an expression has a very high value of description length and a low value of negative logarithm of the probability via a default probability tree, a trained probability tree should assign a larger value of the negative logarithm. In terms of the scatter plot, in that case, a point from the trained tree should be placed to the right and at the same height (because the description length does not vary) of the point coming from the default tree. This is exactly the behaviour that we can appreciate in all of the scatter plots (figs. 2.7, 2.9, 2.11, 2.13 and 2.15). So we can conclude that the probability trees are actually learning.

[5]If we wanted to be extremely precise, the correct term would be an *affine* relationship, since it is of the kind $y = mx + n$. As we will see in the following sentences, we are interested in the idea that both the description length and the negative logarithm grow at the same rate. In simpler terms, a big/low description length should be corresponded with a big/low value in the negative logarithm axis.

Furthermore, if we analyse the files corresponding the stored objects of the trained probability trees (see in appendix A) and consider the probability of the roots of these trees, we realise that the operations that have increased their probability the most are the ones corresponding to the root nodes in the tree decomposition of the used expressions from the Nguyen dataset.

Understanding that the new structure can effectively learn which expressions are more likely to appear for some reference dataset -given enough iterations-, we can perform a more educated navigation through the sample space. In other words, we can also understand the probability tree structure as a method that tries to approximate distributions of closed-form mathematical expressions for symbolic regressions, which would eventually make symbolic regression more efficient.

Chapter 3

Conclusions

The aim of this project was to delve deeper into an open problem in machine learning: approximating the distributions over closed-form mathematical expressions for symbolic regression. Throughout the project, I successfully achieved all the initial objectives. First, I conducted an in-depth study of symbolic regression by proving several key results and reviewing related literature. Second, I thoroughly analysed the functioning of the Bayesian Machine Scientist (BMS) by reading its documentation and experimenting with its code, which provided a comprehensive understanding of its advantages and limitations.

Subsequently, we designed a new probabilistic structure —the probability tree— developed its theoretical foundations, implemented it using an object-oriented approach, and trained it by minimizing the KL divergence, with the goal of improving the efficiency of exploring the space of mathematical models. The results, obtained through the analysis of figs. 2.7, 2.9, 2.11, 2.13 and 2.15, demonstrate that this structure can effectively learn from a given model distribution. In this case, we used expressions generated by the probability tree itself and assigned them a description length using the BMS.

In addition, it is important to offer a more nuanced discussion of what I achieved and what remains open. On the one hand, I demonstrated that the proposed structure —the probability tree— can indeed learn and approximate the model distribution, as evidenced by the change in the description length and the negative logarithm of the probability assigned by the tree before and after training (figs. 2.7, 2.9, 2.11, 2.13 and 2.15). This result is significant, as it shows that the structure can capture the underlying logic of the BMS and therefore holds promise as an efficient tool for exploring the model space in symbolic regression.

Nevertheless, due to the computationally intensive nature of the training process —particularly the minimization of KL divergence over tens of thousands of expressions in each iteration— it was not feasible within the scope of a bachelor's thesis to fully explore the ul-

timate potential of the approximation. Although we observed a clear trend of learning and improvement, the training did not reach full convergence, and the maximum attainable precision of the probability tree remains unknown under the available computational resources and time constraints. Thus, these results should be interpreted as an initial and promising validation, rather than a definitive one. This limitation points to several directions for future research, especially concerning the efficiency, scalability, and generalization capacity of the proposed structure.

Throughout the project, I was able to either learn something new or apply knowledge acquired during my bachelor's degree. First, I had the opportunity to see how Graph Theory -specifically tree graphs- is used to systematically analyse mathematical expressions. While this connection may seem intuitive at first sight, it was surprising and exciting to witness such an elegant application in symbolic regression. In the second place, while studying the theoretical foundations for the BMS, I gained a new perspective on Bayes' rule. Thanks to the Combinatorics and Probability, and Statistics courses from my degree, I found it easier to understand how Bayes' rule is interpreted in machine learning, and what information is conveyed by each of its terms. Furthermore, starting from a problem on wanting to obtain the best function that describes some information I had the fortune of seeing "the magic" of having this problem transformed into one of those on Statistical Mechanics -subject of the degree that I also completed-. Scientific Programming was also pivotal: it enabled me to quickly understand the BMS codebase, which is written in Python, a new language for me at the time. In the third place, thanks to Data Structures and the object-oriented programming skills that I gained there, I was able to properly implement the class of the probability tree, its functions and subsequent structures. Finally, it was thanks to all my academic training that I managed to both design an algorithm to train this structure and to be able to interpret the results in a logical and insightful way.

Overall, this project significantly deepened my understanding of machine learning —a field I am deeply passionate about— and introduced me to symbolic regression, a topic I was initially unfamiliar with. It was also rewarding to see many of the concepts I encountered during this project later appear in the *Machine Learning and Data Mining* course I took during my final semester. I would like to sincerely thank my supervisors, Dr. Marta Sales and Dr. Roger Guimerà, once again, for their patience and dedication throughout my time in their research group. Their guidance made all of this possible.

Finally, since this project is part of a broader, open problem, I would like to humbly propose several directions for future work:

- Progressive deviation reduction when updating node probabilities: To improve KL

Divergence, when updating the node operation probabilities via Gaussian steps, one could start with a high standard deviation and gradually reduce it, thereby limiting randomness over time.

- Hybrid updates: combining random changes and Gaussian steps: Random probability updates (assigning a new value between 0 and 1) can lead to a faster initial decrease in KL Divergence, but they fail to identify which specific operations require adjustment. In contrast, Gaussian steps are slower but more targeted. A promising strategy might be to use random updates in early iterations and switch to Gaussian steps later on.
- Biased node selection: Nodes located higher in the probability tree tend to have a greater impact on the KL Divergence. Therefore, instead of choosing a node completely at random for updates, a weighted selection that favours upper-level nodes could lead to more efficient training.

3.1 Ethics of the Project

Additionally, it is important to state that throughout the development of this project, there was a constant process of self-assessment regarding both its progression and ethical considerations[1]. This included the following elements:

- Equality: recognizing and addressing any potential gender-based discrimination.
- Environmental awareness: identifying the main environmental impacts of the work.
- Social responsibility: acknowledging any social issues and seeking to contribute to their resolution.
- Ethics: adhering to the ethical and deontological codes, demonstrating critical thinking, and making responsible use of the regulations applicable to me as a member of the university community.

Due to the nature of my project, the only directly related component was environmental awareness. In my case, I strived to design efficient, low-energy-consuming algorithms to minimize the environmental footprint as much as possible, without compromising the quality of the experimental results.

[1] They correspond to the competency CT7

Bibliography

[1] Angelis, D., Sofos, F., and Karakasidis, T. E. (2023). Artificial intelligence in physical sciences: Symbolic regression trends and perspectives. *Archives of Computational Methods in Engineering*, 30(6):3845–3865.

[2] Caimo, A. and Friel, N. (2011). Bayesian inference for exponential random graph models. *Social Networks*, 33(1):41–55.

[3] Fajardo-Fontiveros, O., Reichardt, I., De Los Ríos, H. R., Duch, J., Sales-Pardo, M., and Guimerà, R. (2023). Fundamental limits to learning closed-form mathematical models from data. *Nature Communications*, 14(1):1043.

[4] Gerwin, D. (1974). Information processing, data inferences, and scientific generalization. *Behavioral Science*, 19(5):314–325.

[5] Grayeli, A., Sehgal, A., Costilla-Reyes, O., Cranmer, M., and Chaudhuri, S. (2024). Symbolic regression with a learned concept library. In *Advances in Neural Information Processing Systems 37 (NeurIPS 2024)*.

[6] Guimerà, R., Reichardt, I., Aguilar-Mogas, A., Massucci, F. A., Miranda, M., Pallarès, J., and Sales-Pardo, M. (2020). A bayesian machine scientist to aid in the solution of challenging scientific problems. *Science Advances*, 6(5):eaav6971.

[7] Jaynes, E. T. (1957). Information theory and statistical mechanics. *Phys. Rev.*, 106:620–630.

[8] Jin, Y., Fu, W., Kang, J., Guo, J., and Guo, J. (2020). Bayesian symbolic regression.

[9] Langley, P. (1977). Bacon: A production system that discovers empirical laws. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 121–126, Cambridge, MA, USA. William Kaufmann.

[10] Matsubara, Y., Chiba, N., Igarashi, R., and Ushiku, Y. (2024). Rethinking symbolic regression datasets and benchmarks for scientific discovery.

[11] Metropolis, N., Rosenbluth, A. W., Rosenbluth, M. N., Teller, A. H., and Teller, E. (1953). Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, 21(6):1087–1092.

[12] Meurer, A., Smith, C. P., Paprocki, M., Čertík, O., Kirpichev, S. B., Rocklin, M., Kumar, A., Ivanov, S., Moore, J. K., Singh, S., Rathnayake, T., Vig, S., Granger, B. E., Muller, R. P., Bonazzi, F., Gupta, H., Vats, S., Johansson, F., Pedregosa, F., Curry, M. J., Terrel, A. R., Roučka, v., Saboo, A., Fernando, I., Kulal, S., Cimrman, R., and Scopatz, A. (2017). Sympy: symbolic computing in python. *PeerJ Computer Science*, 3:e103.

[13] Nielsen, F. (2020). An elementary introduction to information geometry. *Entropy*, 22(10):1100.

[14] Orzechowski, P., La Cava, W., and Moore, J. H. (2018). Where are we now? a large benchmark study of recent symbolic regression methods. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '18, page 1183–1190, New York, NY, USA. Association for Computing Machinery.

[15] Python Software Foundation (2024). *multiprocessing — Process-based parallelism*. Python documentation, version 3.13.

[16] Velázquez, J. A. R. (2022). Graph theory notes.

[17] Wikipedia contributors (2023). Observed information — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Observed_information&oldid=1182977712. [Online; accessed 17-April-2025].

[18] Wikipedia contributors (2025a). Kullback–leibler divergence — Wikipedia, the free encyclopedia. [Online; accessed 11-April-2025].

[19] Wikipedia contributors (2025b). Metropolis–hastings algorithm — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Metropolis%E2%80%93Hastings_algorithm&oldid=1279572778. [Online; accessed 17-April-2025].

[20] Wikipedia contributors (2025c). Nat (unit) — Wikipedia, the free encyclopedia. [Online; accessed 15-April-2025].

[21] Wikipedia contributors (2025d). Symbolic regression — Wikipedia, the free encyclopedia. [Online; accessed 6-April-2025].

Appendix A

Appendix

A Probability Tree Implementation

The implementation of the probability tree, the scripts used, and the results can be found at https://github.com/LeonelFNR/Probability-Tree-and-BMS. It also contains the files used in the BMS.

B Alternative Strategy for the Probability Tree Training

As said in 2.2.3.1, the current method of randomly updating the probabilities of a node in the tree is through Gaussian steps. This method provides a constant rate of KL Divergence reduction, yet its main drawback is that, for the used configurations, this decrease is slow and not extremely high. Prior to the implementation of this approach, we designed a more random method. This method for updating the probability lists of the nodes consisted of the following steps:

1. From the list containing all functions, variables, and parameters, consider two dictionaries: one with the functions[1] and another one with both the parameters and variables. Each of these dictionaries acts as a (sub) probability list. To obtain the probability list of all the elements one just needs to apply a factor, let's say p, to one dictionary and $1-p$ to the other and add them together. The factor $p \in (0,1)$ just acts as a term that keeps the sum of probabilities to 1.

2. Having selected a random node, select one random function to modify its probability. Then, from the dictionary of probabilities of functions, assign to the chosen function a

[1] In this case I mean the set of operations, like sine, cosine, ...

random value between 0 and 1. Since the operations in the sub-dictionary must sum to one, in this approach, instead of normalizing, we subtract a specific amount from the probability of each operation. This subtracted amount is proportional to the previous probability.

3. Then, the entire probability list is regenerated using a random p factor.

4. If the change reduces the KL Divergence, it is accepted, otherwise rejected and the probability tree went is set back to its previous state.

For this implementation, in addition to training, we performed three more experiments:

- M1: Compare the description lengths and the negative logarithm of the probabilities between the BMS and the probability tree on expressions used in the training. These expressions were sampled from the BMS.

- M2: Compare the description lengths and the negative logarithm of the probabilities between the BMS and the probability tree on expressions not used in the training. These expressions were sampled from the BMS.

- M3: Compare the description lengths and the negative logarithm of the probabilities between the BMS and the probability tree on expressions not used in the training. These expressions were sampled from the probability tree.

It is important to note that since these experiments were conducted prior to implementing the Gaussian steps method, the training datasets are different. That's why the initial values of the KL Divergence are different. However, what is interesting here is the relative decrease in the value of the parameter. The results of these experiments can be appreciated in the following figures.

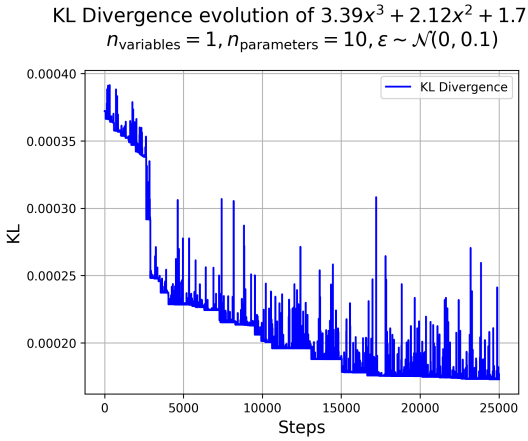


Figure A.1: KL Divergence evolution of $3.39x^3 + 2.12x^2 + 1.78x$

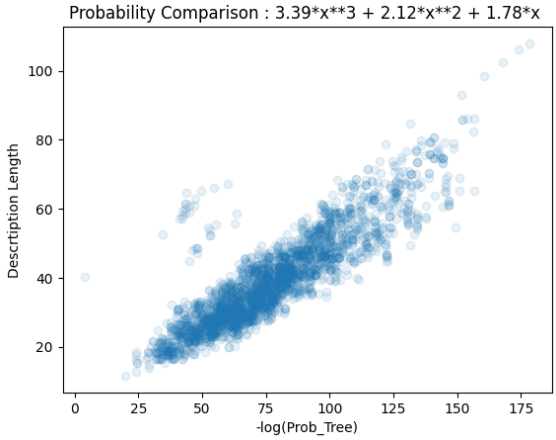


Figure A.2: M1 results for $3.39x^3+2.12x^2+1.78x$

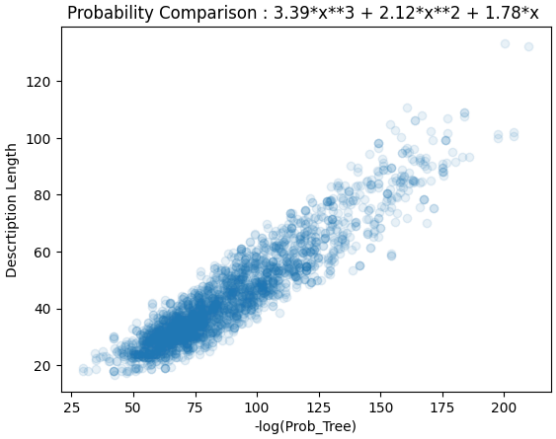


Figure A.3: M2 results for $3.39x^3+2.12x^2+1.78x$

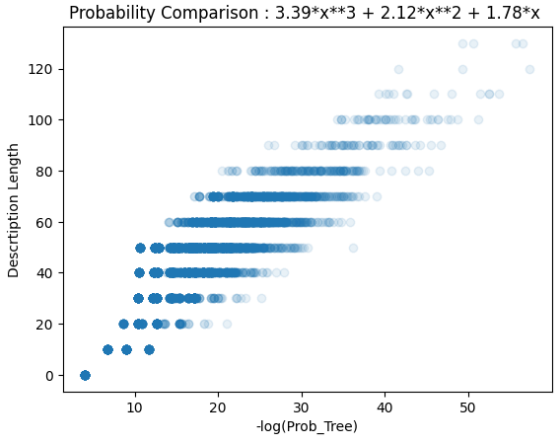


Figure A.4: M3 results for $3.39x^3+2.12x^2+1.78x$

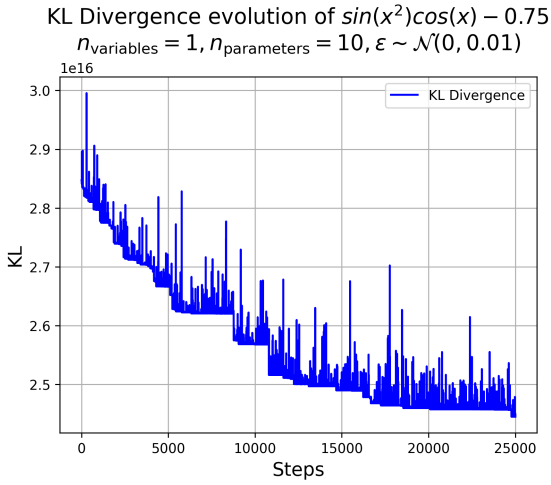


Figure A.5: KL Divergence evolution of $\sin(x^2)\cos(x) - 0.75$

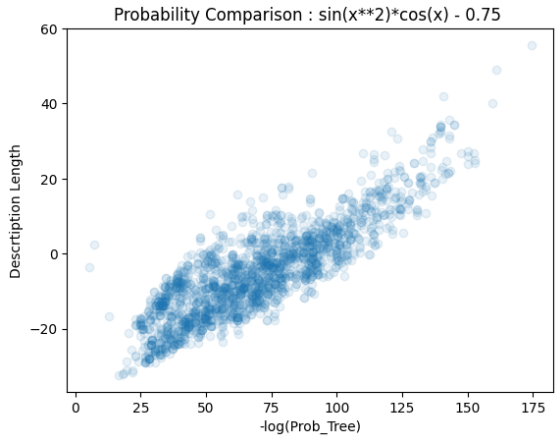


Figure A.6: M1 results for $\sin(x^2)\cos(x) - 0.75$

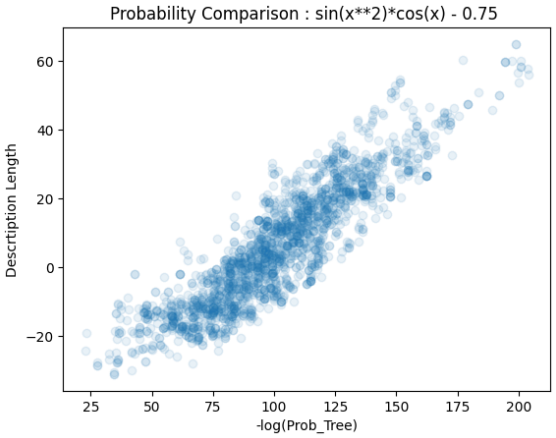


Figure A.7: M2 results for $\sin(x^2)\cos(x) - 0.75$

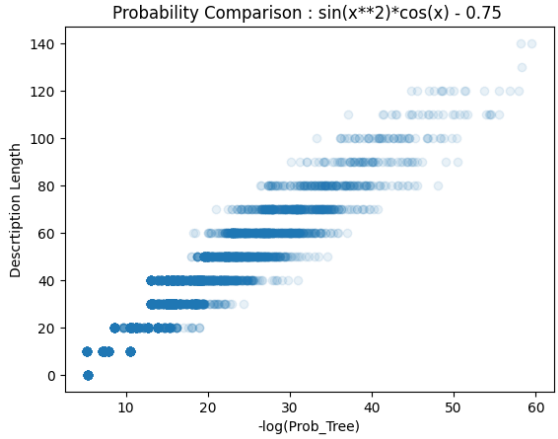


Figure A.8: M3 results for $\sin(x^2)\cos(x) - 0.75$

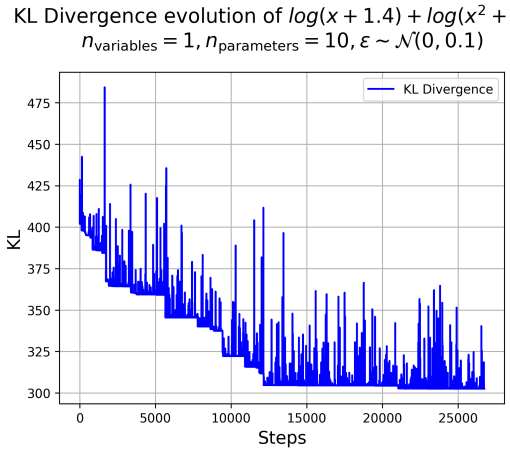


Figure A.9: KL Divergence evolution of $\log(x + 1.4) + \log(x^2 + 1.3)$

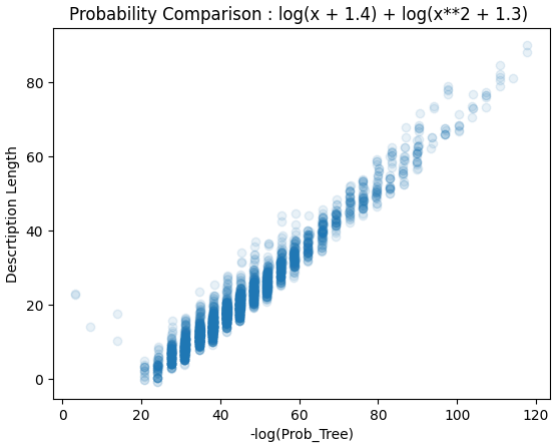


Figure A.10: M1 results for $\log(x + 1.4) + \log(x^2 + 1.3)$

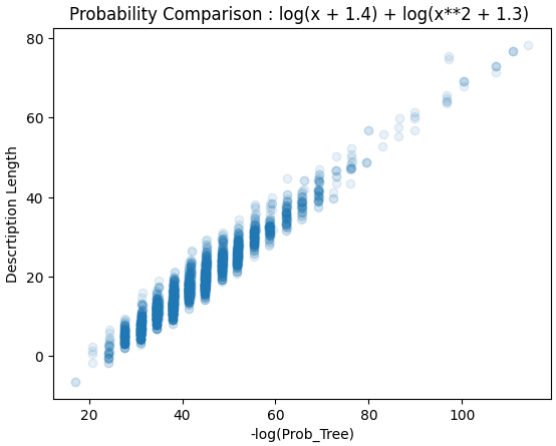


Figure A.11: M2 results for $\log(x + 1.4) + \log(x^2 + 1.3)$

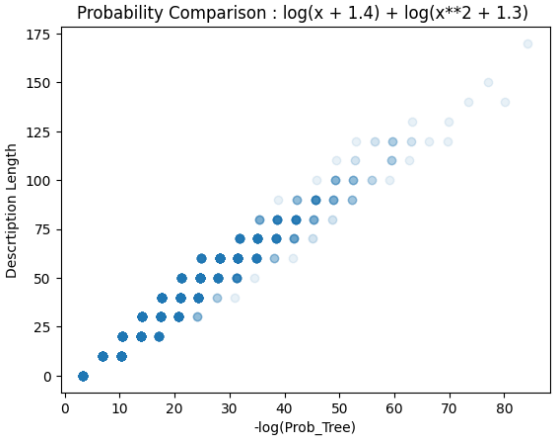


Figure A.12: M3 results for $\log(x + 1.4) + \log(x^2 + 1.3)$

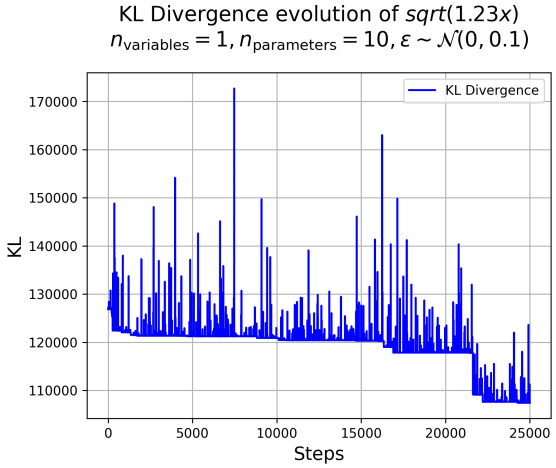


Figure A.13: KL Divergence evolution of $\sqrt{1.23x}$

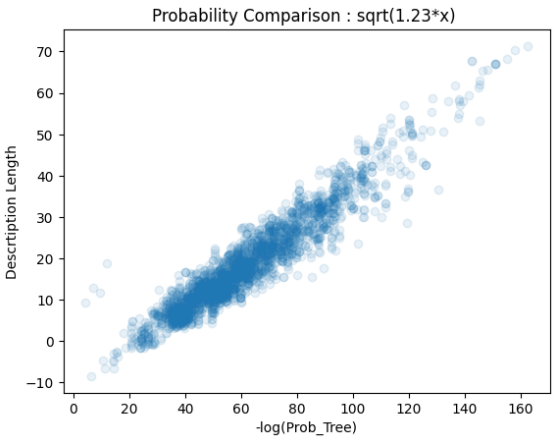


Figure A.14: M1 results for $\sqrt{1.23x}$

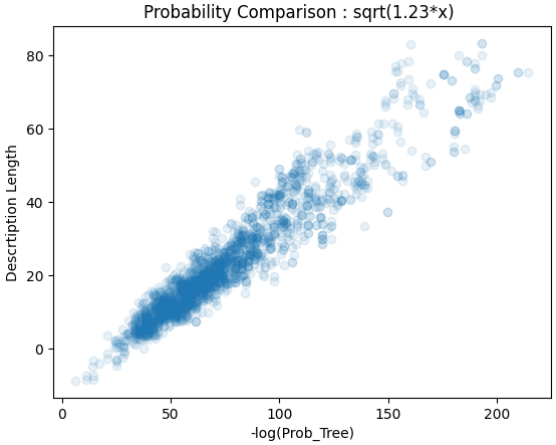


Figure A.15: M2 results for $\sqrt{1.23x}$

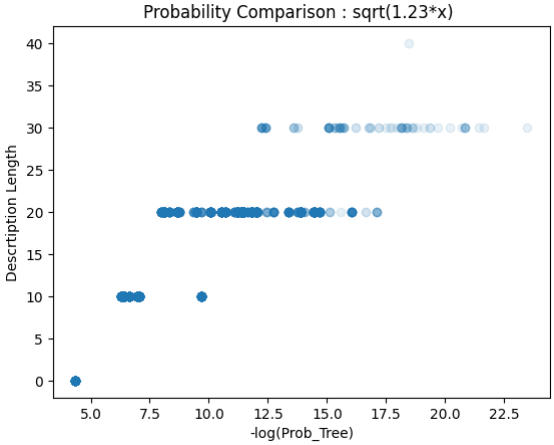


Figure A.16: M3 results for $\sqrt{1.23x}$

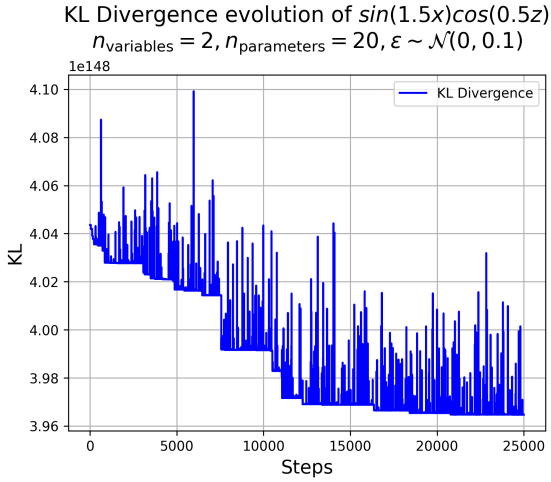


Figure A.17: KL Divergence evolution of $\sin(1.5x)\cos(0.5)$

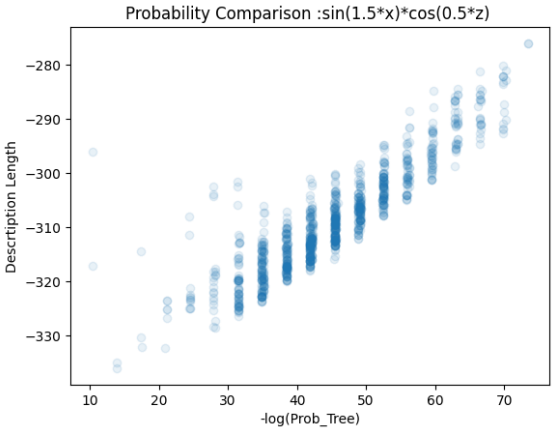


Figure A.18: M1 results for $\sin(1.5x)\cos(0.5)$

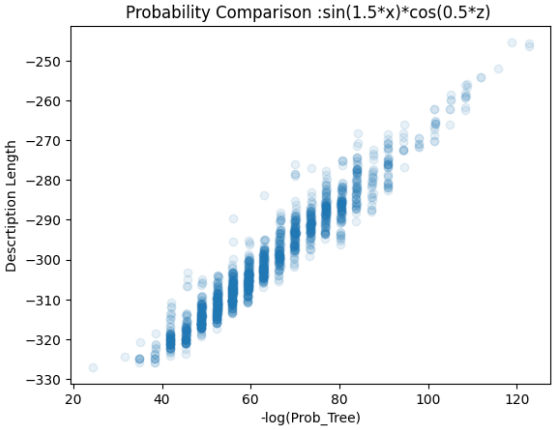


Figure A.19: M2 results for $\sin(1.5x)\cos(0.5)$

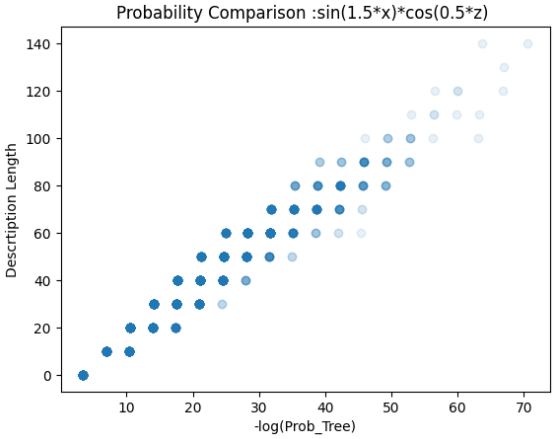


Figure A.20: M3 results for $\sin(1.5x)\cos(0.5)$

Inspecting the KL Evolution plots, we first notice that they present many "spikes", sudden increases. In this case, this makes sense if we take into account that with this procedure the random changes are very abrupt, because a probability can take any value between 0 and 1. This leads to a really interesting behaviour: less changes are accepted, but in some step there is one change that makes the KL Divergence to drop. Thanks to this effect, the relative decrease in KL Divergence is higher (with respect to the number of iterations) than when using the Gaussian steps.

Furthermore, inspecting the plots corresponding to experiments M1, M2, M3 the results look promising, considering the reasoning stated in 2.2.5, because in all cases the relationship between the description lengths and the negative logarithm of the probabilities is linear, with

slope generally close to one.

Nevertheless, when I inspected the probability trees corresponding to these experiments, the probability lists[2] did not resemble the binary tree representation of the expressions in the Nguyen dataset. In the following figures, I provide two examples of this behaviour and I compare them with the results using Gaussian steps.

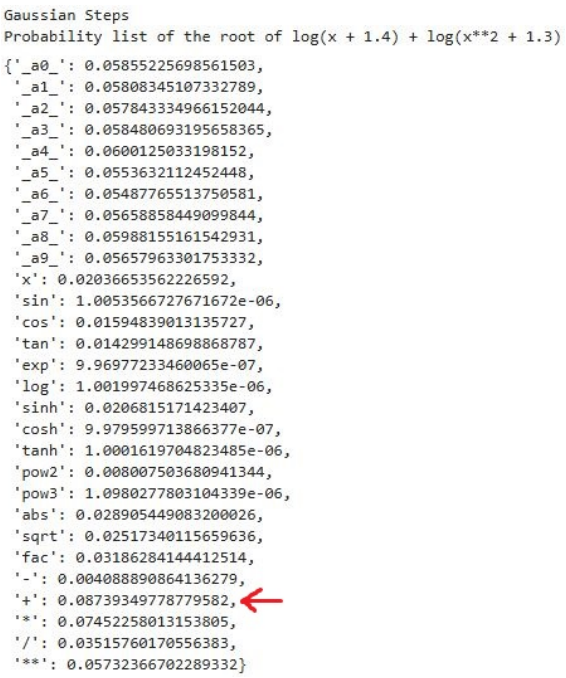


Figure A.21: Root probability list of the Gaussian step trained tree of $\log(x + 1.4) + \log(x^2 + 1.3)$

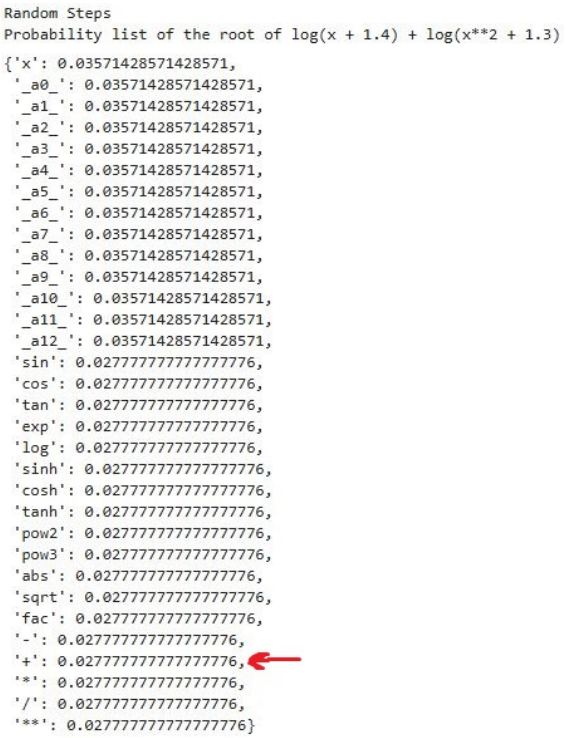


Figure A.22: Root probability list of the random step trained tree of $\log(x + 1.4) + \log(x^2 + 1.3)$

In this case, if we were to decompose via a binary tree the function $\log(x + 1.4) + \log(x^2 + 1.3)$, the root would be the $+$ sign. In the case of the Gaussian step trained probability tree, we can see that the highest probability is assigned to this exact operation and, in general, the operations that are unlikely to be the root have their probabilities assigned to very low values. The probability list of the root of the Random steps trained tree appears unchanged: no change in this probability list was accepted because of the reasons explained before.

[2] The probability list that was mainly inspected was the one corresponding to the root, which is the node that will appear in all the expressions. If the probability list of the node is not as expected, there is not much hope for the rest of the nodes.

```
Gaussian Steps
Probability list of the root of sqrt(1.23*x)
{'_a0_': 0.06609698062693269,
 '_a1_': 0.06688417891159128,
 '_a2_': 0.06342209122452815,
 '_a3_': 0.06349070945707783,
 '_a4_': 0.07000458155462566,
 '_a5_': 0.07168384250854719,
 '_a6_': 0.0671279425974913,
 '_a7_': 0.06479782776941428,
 '_a8_': 0.06920410486466406,
 '_a9_': 0.062059291268398535,
 'x': 0.026867548647667126,
 'sin': 0.008415364515613147,
 'cos': 1.0361153674839403e-06,
 'tan': 0.011944249974943205,
 'exp': 1.0650158675193464e-06,
 'log': 1.3367754350021491e-06,
 'sinh': 0.0105188105930832,
 'cosh': 0.029965416703148624,
 'tanh': 1.0082021306546616e-06,
 'pow2': 0.006114873128190762,
 'pow3': 1.048619589540916e-06,
 'abs': 1.2659532189447115e-06,
 'sqrt': 0.08076695152182244,
 'fac': 0.012960154883468902,
 '-': 1.0012648219852529e-06,
 '+': 0.008573719781089858,
 '*': 0.06204128973033253,
 '/': 0.009050262586637985,
 '**': 0.0680020452043002}
```

Figure A.23: Root probability list of the Gaussian step trained tree of $\sqrt{1.23x}$

```
Random Steps
Probability list of the root of sqrt(1.23*x)
{'x': 0.012784697526432876,
 '_a0_': 0.012784697526432876,
 '_a1_': 0.012784697526432876,
 '_a2_': 0.012784697526432876,
 '_a3_': 0.012784697526432876,
 '_a4_': 0.012784697526432876,
 '_a5_': 0.012784697526432876,
 '_a6_': 0.012784697526432876,
 '_a7_': 0.012784697526432876,
 '_a8_': 0.012784697526432876,
 '_a9_': 0.012784697526432876,
 '_a10_': 0.012784697526432876,
 '_a11_': 0.012784697526432876,
 '_a12_': 0.012784697526432876,
 'sin': 0.018656885060586742,
 'cos': 0.018656885060586742,
 'tan': 0.018656885060586742,
 'exp': 0.018656885060586742,
 'log': 0.018656885060586742,
 'sinh': 0.017822774666686924,
 'cosh': 0.018656885060586742,
 'tanh': 0.018656885060586742,
 'pow2': 0.027142339487276082,
 'pow3': 0.018656885060586742,
 'abs': 0.03694579427628066,
 'sqrt': 0.018656885060586742,
 'fac': 0.018656885060586742,
 '-': 0.018656885060586742,
 '+': 0.018656885060586742,
 '*': 0.018656885060586742,
 '/': 0.018656885060586742,
 '**': 0.47790693535148177}
```

Figure A.24: Root probability list of the random step trained tree of $\sqrt{1.23x}$

In this other example, if we decompose the function $\sqrt{1.23x}$, the root would be the $\sqrt{}$ sign. In the case of the Gaussian step trained probability tree, we can see that the highest probability is assigned in fact to this operation and that, the majority of the other operations have had their probabilities lowered. However, even though the probability list of the root of the Random steps trained tree had several changes in the probabilities of its operations, the square root did not benefit from these changes, having one of the lowest probabilities. This second result can be understood as follows: when training the second tree, if a random change decreased the probability of an operation it should not, but also decreased the probabilities of other incorrect operations, that change may have been accepted, leading to a decrease in the KL Divergence, but an incorrect characterization of the probability space.

Given these counterintuitive results, after careful thought and consideration by both my

tutors and myself, we concluded that the tree was mainly learning from the given prior distribution, which is located in the file system for the BMS. This accounts for both the very accurate linear relationships between the description lengths and the negative logarithms of the probabilities, and the incorrect characterization of the probability distributions. Since the BMS takes into account the prior distribution for sampling expressions, we were training the tree to learn from the priors themselves. Moreover, there is an excessive randomness in the procedure of updating probabilities, which negatively affects progress toward finding the appropriate probability distribution and, despite allowing a rapid decrease in KL Divergence, it gets stuck at some point.